VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra aplikované matematiky

Multifrontální metoda pro řešení rozsáhlých řídkých soustav lineárních rovnic

Multifrontal method for solution of large sparse systems of linear equations

2017 Bc. Lukáš Mihula

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra aplikované matematiky

Zadání diplomové práce

Student: **Bc. Lukáš Mihula**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 1103T031 Výpočetní matematika

Téma: Multifrontální metoda pro řešení rozsáhlých řídkých soustav lineárních rovnic
Multifrontal method for solution of large sparse systems of linear equations

Jazyk vypracování: čeština

Zásady pro vypracování:

Řešení soustav lineárních rovnic je jednou z nejčastějších úloh numerické matematiky. Rozsáhlé inženýrské úlohy popsané parciálními diferenciálními rovnicemi jsou nejčastěji diskretizovány pomocí metody konečných prvků. Vznikají tak soustavy s velmi velkou a řídkou systémovou maticí se specifickou strukturou nenulových prvků. V zásadě existují dva základní přístupy, jak tyto soustavy řešit: iterační a přímé metody. Iterační metody (např. metoda sdružených gradientů) jsou jednodušší na implementaci vč. paralelní. Přímé metody jsou založeny na maticových rozkladech, jež úlohu převádějí na řešení soustavy s trojúhelníkovou maticí. Přímé metody mají řadu výhod: zejména vysokou přesnost řešení, numerickou stabilitu a robustnost vůči špatně podmíněným maticím. Mají však mnohem vyšší paměťovou náročnost a paralelismus je omezen. Proto lze s nimi řešit jen podstatně menší úlohy. Jejich efektivní implementace je podstatně složitější. Je-li matice symetrická, což platí pro významnou množinu problémů, lze k řešení soustavy využít Choleského rozklad. Existují jeho specifické varianty pro velké řídké matice. V této oblasti již existují velmi sofistikované přístupy. K nim patří i multifrontální metoda, využívající k řešení řídké soustavy operace s malými hustými podmaticemi. Právě na tuto metodu by práce měla být primárně zaměřena.

Cíle práce:
1. nastudování a shrnutí významných existujících přístupů pro přímé řešení rozsáhlých řídkých soustav,
2. podrobné nastudování a shrnutí multifrontální metody,
3. implementace multifrontální metody objektově orientovaným způsobem v C++ se zřetelem na didaktický, přehledný kód,
4. diskuze nejvýznamnějších problémů, které při této implementaci vyvstávají,
5. otestování implementace na vybraných benchmarcích a diskuze výkonnostních aspektů,
6. nastudování existujících široce používaných implementací a srovnání s vlastní implementací.

Seznam doporučené odborné literatury:

Podle doporučení školitele.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Václav Hapla**

Datum zadání: 01.09.2016

Datum odevzdání: 28.04.2017

doc. RNDr. Jiří Bouchala, Ph.D.
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 28. dubna 2017

.....

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 28. dubna 2017 .....

Abstrakt

Řešení rozsáhlých řídkých soustav lineárních rovnic patří mezi nejčastější úkony numerické matematiky. K řešení takovýchto soustav v podstatě existují dva základní přístupy: přímé a iterační metody. Principem iteračních metod je postupné zpřesňování odhadu řešení. Přímé metody využívají faktorizaci systémové matice soustavy, která převede úlohu na řešení soustavy s trojúhelníkovou maticí. V této práci je popsána přímá multifrontální metoda a její praktická implementace v jazyce C++.

Klíčová slova: Multifrontální metoda, Choleského faktorizace, LU rozklad, řídká matice, soustava lineárních rovnic, C++

Abstract

The solution of large sparse systems of linear equations is among the most frequent tasks of numerical mathematics. To solve such systems, there are basically two basic approaches: direct and iterative methods. The principle of iterative methods is the gradual refinement of the estimate of solution. Direct methods use the factorization of the system matrix, which transforms the task into solving a system with triangular matrix. This paper describes a direct multifrontal method and its practical implementation in C++ language.

Key Words: Multifrontal method, Cholesky factorization, LU decomposition, sparse matrix, system of linear equations, C++

Obsah

Seznam použitých zkratek a symbolů

BLAS	–	Basic Linear Algebra Subprograms
COO	–	Coordinate
CSC	–	Compressed Sparse Columns
CSR	–	Compressed Sparse Rows
IDE	–	Integrated Development Environment
MUMPS	–	Multifrontal Massively Parallel sparse direct Solver

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Hlavním úkolem diplomové práce bylo seznámení se s multifrontální metodou a její následná implementace. Inženýrské úlohy dnešní doby jsou popsány parciálními diferenciálními rovnicemi, které jsou nejčastěji diskretizovány pomocí metody konečných prvků. Diskretizací vznikají rozsáhlé řídké soustavy lineárních rovnic. Jednou možností, jak tyto soustavy řešit, jsou přímé metody, mezi které spadá multifrontální metoda. Přímé metody jsou založeny na faktorizaci systémové matice, která převede zadaný problém na řešení soustavy s trojúhelníkovou maticí.

První část této práce se věnuje detailnímu popisu multifrontální metody. Seznámíme se s důležitými pojmy jako eliminační strom, frontální a update matice, supernody ad. I když jsou tyto pojmy probrány v rámci multifrontální metody, patří mezi základní pilíře přímých solverů. Po teoretickém základě pokračuje část, ve které je podrobně rozebrána praktická implementace multifrontální metody v programovacím jazyce C++. V této části jsou diskutovány problémy, které při implementaci vyvstaly a popsány programátorské techniky, které byly při vytváření aplikace použity. Čtenář je seznámen s kompletní strukturou aplikace, ukázkou programu a výstupními daty.

Druhá část je zaměřena na testování vytvořené implementace. Porovnává se výkon aplikace na dvou počítačích a jsou vysloveny závěry vyplývající ze zpracování dat, které byly k testování použity.

V závěrečné kapitole jsou představeny již existující přímé solvery. Je popsáno základní použití těchto solverů a v lehké míře zmíněna jejich interní funkcionalita.

2 Úvodní poznámky

V tomto textu se setkáme s několika (i elementárními) pojmy, jejichž znalost je ovšem nutná ke správnému pochopení jednotlivých kapitol. Pro čtenáře je přívětivější, pokud je daný text co nejvíce samostatný a proto máme k dispozici tuto kapitolu, která slouží k upřesnění některých pojmů převážně z lineární algebry a teorie grafů. Stále se ale předpokládá, že čtenář má základní znalost vysokoškolské matematiky.

2.1 Základní pojmy

Řídká matice

Řídká matice představuje speciální typ matic, které mají většinu svých prvků nulových. Nenulové prvky se pak nejčastěji nacházejí kolem hlavní diagonály matice, ale není to nutnou podmínkou. Rozložení nenulových prvků v řídké matici má vliv na náročnost výpočtu Gaussovy eliminace, LU či Choleského rozkladu a dalších metod. Tyto metody pak často nejdou na řídké matice efektivně aplikovat a proto existují jejich alternativní „řídké“ verze.

$$\begin{pmatrix} 6 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 \\ 2 & 0 & 5 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

Obrázek 1: Ukázka řídké matice

Graf

Graf G je uspořádanou dvojicí neprázdné množiny vrcholů V a množiny E dvouprvkových podmnožin množiny V (hran).

Cyklus

Cyklus (kružnice) je posloupnost vrcholů a hran $(v_0, e_1, v_1, \ldots, e_k, v_k = v_0)$, kde vrcholy $(v_0, \ldots, v_{k-1})$ jsou navzájem různé vrcholy v daném grafu.

Strom

Strom je souvislý graf, který neobsahuje žádné cykly (je acyklický). Vrcholy stromu se označují uzly. Uzly, které nemají žádného potomka, se označují jako listy. Uzel, který nemá žádného rodiče, se označuje jako kořen. Toto označení určuje orientaci stromu - z grafového hlediska není mezi kořenem a listem rozdíl. Nesouvislý acyklický graf se nazývá les (skládá se z více stromů).

Redukovatelná matice

Čtvercová $n \times n$ matice A je redukovatelná, pokud lze indexy $1, 2, .., n$ rozdělit do dvou

neprázdných disjunktních množin $i_1, i_2, \ldots, i_u$ a $j_1, j_2, \ldots, j_v$ $(u + v = n)$ tak, že

$$a_{i_\alpha, j_\beta} = 0, \quad \alpha = 1, 2, \ldots, u, \quad \beta = 1, 2, \ldots, v.$$

Pokud čtvercová matice není redukovatelná, označuje se jako neredukovatelná.

Pozitivně definitní matice

Reálná $n \times n$ matice A se nazývá pozitivně definitní, pokud pro každý nenulový vektor $x \in R^n$ platí

$$x^T A x > 0.$$

Hlavní minor matice

Hlavní minor $|A_i|$ $n \times n$ matice A je determinant matice $i \times i$, která byla vytvořena z prvních i řádků a sloupců matice A. Symetrická pozitivně definitní matice má všechny hlavní minory kladné.

Ortogonální matice

Reálná čtvercová matice A se nazývá ortogonální, pokud transponovaná matice A^T je současně inverzní maticí. Platí tedy

$$A^T = A^{-1}.$$

2.2 Řešení lineárních soustav pomocí faktorizace

Uvažujme obecnou faktorizaci čtvercové matice A do tvaru

$$A = LU,$$

kde matice L má dolní trojúhelníkový tvar a matice U horní trojúhelníkový tvar. Rozklad matice do takového tvaru lze efektivně využít k nalezení řešení systému lineárních rovnic. Uvažujme systém ve tvaru

$$Ax = b,$$

kde A představuje koeficienty soustavy (jedná se o tzv. systémovou matici), x je vektor neznámých a b vektor pravé strany. Po faktorizaci matice A je originální systém převeden do tvaru

$$LUx = b.$$

Řešení takto zadaného systému můžeme rozdělit do dvou kroků. Nejprve nalezněme y, pro které platí

$$Ly = b.$$

Tento krok se označuje jako dopředná substituce. Po nalezení vektoru y pak hledáme x, pro které platí

$$Ux = y.$$

Tento krok se označuje jako zpětná substituce a po nalezení vektoru x získáváme řešení originálního systému.

Dalším typem faktorizace je rozložení čtvercové matice A do tvaru

$$A = QR,$$

kde Q je ortogonální matice a R je horní trojúhelníková matice. Originální systém je pak převeden do tvaru

$$QRx = b,$$

který je upraven do tvaru

$$Rx = Q^T b.$$

Takto upravený systém je vyřešen pomocí zpětné substituce.

3 Multifrontální metoda

Multifrontální metoda patří mezi přímé metody pro řešení řídkých systémů lineárních rovnic. Její tvůrci jsou Iain Duff a John Reid, kteří tuto metodu vyvinuli v roce 1983. Pojem „multifrontální“ použili Duff a Reid, protože metodu vyvíjeli jako zobecnění frontální metody, kterou vytvořil Bruce Irons. Multifrontální metoda uspořádává celkovou faktorizaci jedné velké řídké matice do posloupnosti částečných faktorizací menších hustých (plných) matic.

Za léta používání se multifrontální metoda prokázala jako velice přínosná. Nasvědčuje tomu její využití v mnoha vědeckých a inženýrských aplikacích. Metoda byla využita např. k vyřešení Navier-Stokesovy rovnice pro výpočet dynamiky kapalin, při simulaci polovodičových zařízení a pro řešení problémů oddělitelné optimalizace. Metoda se také ukázala jako vhodný nástroj lineárního programování pro efektivní řešení řídkého lineárního problému nejmenších čtverců, který vyvstal z Karmarkarovy metody [1].

Multifrontální metodu lze aplikovat na symetrické (Choleského faktorizace) i nesymetrické (LU rozklad) matice. V tomto textu se zaměříme převážně na symetrické matice, včetně samotné implementace multifrontální metody, ale bude zmíněna i verze pro nesymetrické matice.

3.1 Choleského faktorizace

Faktorizace čtvercové symetrické pozitivně definitní matice A do tvaru $A = LL^T$ lze vykonat několika způsoby. Pokud provádíme faktorizaci po řádcích, je v každém kroku vyřešením trojúhelníkového systému faktorizován právě jeden řádek. Při sloupcové faktorizaci je v jednom kroku spočten daný sloupec aplikací příspěvků ze všech předchozích sloupců a následným škálováním. Posledním způsobem je faktorizace po submaticích, kdy se v jednotlivých krocích faktorizuje sloupec a jsou spočteny všechny příspěvky do submatice zbývající ke zpracování. Na základě tohoto rozdělení můžeme na multifrontální metodu nahlížet jako na řídkou Choleského faktorizaci po submaticích.

Jeden krok faktorizace husté $n \times n$ matice A pomocí submaticového Choleského přístupu můžeme zapsat jako

$$A = \begin{pmatrix} d & v^T \\ v & C \end{pmatrix} = \begin{pmatrix} \sqrt{d} & 0 \\ v/\sqrt{d} & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & C - vv^t/d \end{pmatrix} \begin{pmatrix} \sqrt{d} & v^t/\sqrt{d} \\ 0 & I \end{pmatrix}, \tag{1}$$

kde d je první diagonální element a v je vektor o velikosti $(n-1)$. Submatice $C - vv^t/d$ je část zbývají k faktorizaci. Faktorizace zbývající části může být vykonána rekurzivně stejným způsobem.

Jeden krok faktorizace lze vyjádřit v blokovém tvaru. Předpokládejme, že bylo provedeno i kroků faktorizace ($i > 1$). Je výhodné zapsat tuto částečnou faktorizaci pomocí 2×2 blokového

rozdělení:

$$A = \begin{pmatrix} B & V^T \\ V & C \end{pmatrix} = \begin{pmatrix} L_B & 0 \\ VL_B^{-t} & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & C - VB^{-1}V^t \end{pmatrix} \begin{pmatrix} L_B^t & L_B^{-1}V^t \\ O & I \end{pmatrix},$$

kde $B = L_B L_b^t$ je částečná Choleského faktorizace $(i-1) \times (i-1)$ submatice B. Tato matice se označuje jako hlavní vedoucí submatice. Submatice $C - VB^{-1}V^t$ se označuje jako Schurův doplněk a reprezentuje část matice A zbývající ke zpracování.

Je vhodné uvědomit si, že submatice $-VB^{-1}V^t$ o velikosti $(n-i+1) \times (n-i+1)$ reprezentuje veškeré příspěvky z prvních $i-1$ sloupců a řádků do submatice C. Tato matice příspěvků může být zapsána ve smyslu prvních $i-1$ sloupců Choleského faktorizace:

$$-VB^{-1}V^t = -(VL_B^{-t})(L_B^{-1}V^t) = -\sum_{k=1}^{i-1} \begin{pmatrix} l_{i,k} \\ \vdots \\ l_{n,k} \end{pmatrix} \begin{pmatrix} l_{i,k} & \dots & l_{n,k} \end{pmatrix}.$$

Jedná se tedy o sumu vnějších součinů příspěvků jednotlivých $i-1$ sloupců Choleského faktorizace. Tento zápis vnějších součinů příspěvků je základem pro definici update a frontálních matic, které se využívají u řídké faktorizace. Na multifrontální metodu lze nahlížet jako na efektivní způsob spravování vnějších součinů příspěvků (pokud je faktorizovaná matice řídká).

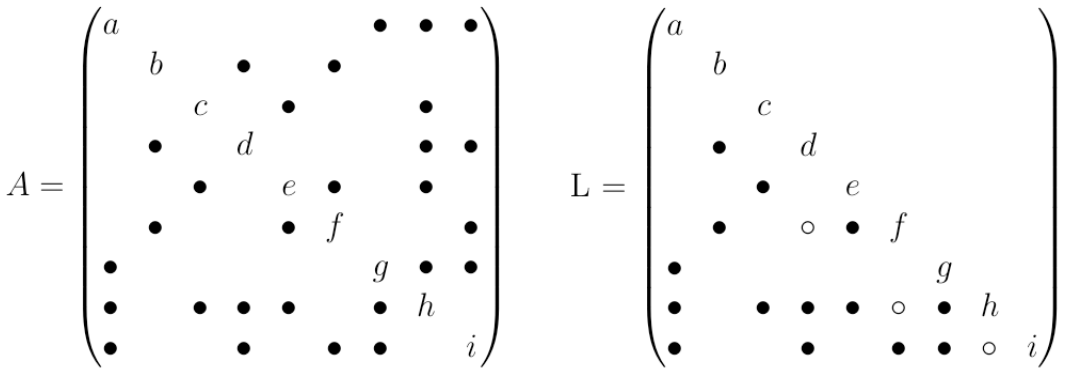
3.2 Eliminační strom

Jak již bylo řečeno, multifrontální metoda efektivně pracuje s příspěvky jednotlivých sloupců Choleského faktorizace pomocí update a frontálních matic. Tyto matice obsahují sumu nějaké podmnožiny vnějších součinů příspěvků. Tuto podmnožinu můžeme vnímat jako množinu příspěvků, které „dávají smysl“. V řídké matici díky velkému množství nulových prvků často nedochází k ovlivnění výpočtu předchozím sloupcem a proto je neefektivní snažit se zjišťovat příspěvky, které nejsou v daném kroku využity, popř. ani neexistují (jsou nulové). Z toho důvodu byla vytvořena struktura udávající příspěvky, které mají být do dané podmnožiny sumy vnějších součinů přidány. Tato struktura se nazývá eliminační strom.

Mějme $n \times n$ řídkou symetrickou pozitivně definitní matici A s Choleského faktorem L (obrázek 2). Eliminační strom matice A je definován jako struktura s n uzly $\{1, \dots, n\}$ tak, že uzel p je rodičem uzlu j, pouze pokud platí

$$p = \min \{i > j | l_{ij} \neq 0\}.$$

Pokud je A neredukovatelná, lze jednoduše ověřit, že se jedná o strom. Pokud je A redukovatelná, výslednou strukturou je les. Pomocí písmene j označujeme sloupec v dané matici a jeho odpovídající uzel v eliminačním stromě.



Obrázek 2: Řídká matice A a její Choleského faktor L

Pro označení eliminačního stromu matice A použijeme $T(A)$. Eliminační strom je důležitou strukturou multifrontální metody a hraje velkou roli ve faktorizaci řídkých matic. Nyní zmíníme několik významných vlastností eliminačního stromu.

Věta 1 *Pokud je uzel k potomkem uzlu j v eliminačním stromě, pak je struktura vektoru $(l_{j,k}, \ldots, l_{n,k})^t$ obsažena ve struktuře vektoru $(l_{j,j}, \ldots, l_{n,j})^t$. Strukturou vektoru (sloupce) myslíme množinu řádkových indexů nenulových prvků v daném sloupci.*

Věta 2 *Pokud platí $l_{j,k} \neq 0$ pro $k < j$, pak je uzel k potomkem uzlu j v eliminačním stromě.*

Pro označení množiny všech potomků uzlu j (včetně uzlu j) použijeme $T[j]$. Plné černé puntíky na obrázku 2 reprezentují originální nenulové prvky v matici A. Prázdné puntíky se označují jako *fill ins* (doplňky) a reprezentují nové nenulové prvky matice L, které vznikly faktorizací.

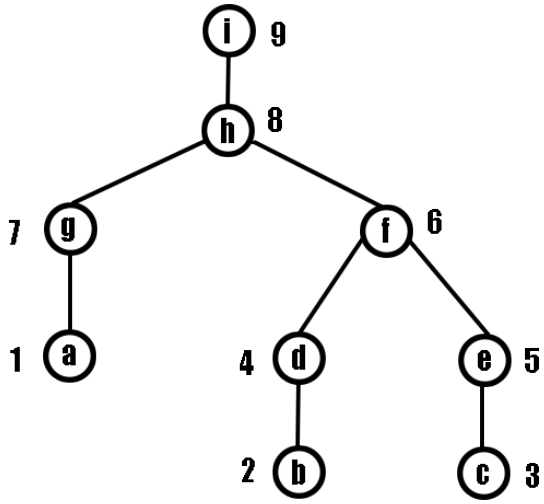
Na obrázku 3 je příklad eliminačního stromu matice A. Z eliminačního stromu můžeme jednoduše vyčíst, že $T[6] = \{2, 3, 4, 5, 6\}$. Nyní víme co je eliminační strom a k čemu se používá, můžeme se proto přesunout na popis dalších důležitých pojmů multifrontální metody - frontální a update matice.

3.3 Frontální a update matice

Mějme $n \times n$ matici A a její Choleského faktor L. Uvažujme nyní j-tý sloupec matice L. Nechť $i_0, i_1, ..., i_r$ jsou řádkové indexy nenulových prvků v $L_{*,j}$, kdy $i_0 = j$, tedy sloupec j má r nenulových prvků.

Subtree (podstromová) update matice náležící sloupci j dané matice A je definována jako

$$\overline{U}_j = - \sum_{k \in T[j]-\{j\}} \begin{pmatrix} l_{j,k} \\ l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{pmatrix} \begin{pmatrix} l_{j,k} & l_{i_1,k} & \ldots & l_{i_r,k} \end{pmatrix}. \tag{2}$$



Obrázek 3: Ukázka eliminačního stromu matice A z obrázku 2

Subtree update matice tedy obsahuje veškeré vnější součiny příspěvků z předešlých sloupců, které jsou potomkem uzlu j v eliminačním stromě. Na základě věty 1 platí, že pokud je k potomkem j v eliminačním stromě, pak množina řádkových indexů nenulových prvků $(l_{j,k}, l_{j+1,k}, \ldots, l_{n,k})^t$ je obsažena v $\{j, i_1, \ldots, i_r\}$.

Frontální matice F_j j-tého sloupce dané matice A je definována jako

$$F_j = \begin{pmatrix} a_{j,j} & a_{j,i_1} & \ldots & a_{j,i_r} \\ a_{i_1,j} & & & \\ \vdots & & 0 & \\ a_{i_r,j} & & & \end{pmatrix} + \overline{U}_j. \tag{3}$$

Jelikož diagonální prvek subtree update i frontální matice spadá mezi nenulové prvky daného sloupce, je řád $r+1$ obou těchto matic zároveň celkovým počtem nenulových prvků j-tého sloupce faktoru L.

Podívejme se nyní přesněji na vytvoření matice $\overline{U}_j$. Díky věty 2 můžeme $\overline{U}_j$ vyjádřit pomocí dvou komponent. Všechna $k < j$ s $l_{j,k} \neq 0$ patří do množiny potomků j-tého sloupce a jejich vnější součiny příspěvků jsou obsaženy v matici $\overline{U}_j$. Zápis pomocí dvou komponent vypadá následovně:

$$\overline{U}_j = -\sum_{\substack{k<j \\ l_{j,k}\neq 0}} \begin{pmatrix} l_{j,k} \\ l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{pmatrix} \begin{pmatrix} l_{j,k} & l_{i_1,k} & \ldots & l_{i_r,k} \end{pmatrix} - \sum_{\substack{k\in T[j]-\{j\} \\ l_{j,k}=0}} \begin{pmatrix} 0 \\ l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{pmatrix} \begin{pmatrix} 0 & l_{i_1,k} & \ldots & l_{i_r,k} \end{pmatrix}.$$

Z této rovnice můžeme vidět, že pouze první z komponent přispívá do prvního sloupce matice

$\overline{U}_j$. První sloupec matice $\overline{U}_j$ proto můžeme zapsat jako

$$- \sum_{\substack{k<j \\ l_{j,k}\neq 0}} l_{j,k} \begin{pmatrix} l_{j,k} \\ l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{pmatrix}.$$

Tento sloupec obsahuje veškeré příspěvky do sloupce j a označuje se jako úplný sloupcový update sloupce j.

Jak můžeme vidět v (3), první řádek a sloupec matice F_j je tvořen příslušným řádkem a sloupcem matice A a úplným sloupcovým updatem sloupce j. Z toho vyplývá, že při výpočtu frontální matice F_j je první řádek a sloupec této matice plně aktualizován. Můžeme tedy provést jeden krok eliminace (1) a získat tak nenulové prvky faktorového sloupce $L_{*,j}$. Přesněji máme

$$F_j = \begin{pmatrix} l_{j,j} & 0 \\ l_{i_1,j} & \\ \vdots & I \\ l_{i_r,j} & \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & U_j \end{pmatrix} \begin{pmatrix} l_{j,j} & l_{i_1,j} \dots l_{i_r,j} \\ 0 & I \end{pmatrix}.$$

Jelikož jsou $j, i_1, \dots, i_r$ indexy nenulových prvků ve sloupci $L_{*,j}$ je vektor $(l_{j,j}, l_{i_1,j}, \dots, l_{i_r,j})^t$ hustý (plný). To může nastat pouze pokud je první sloupec a řádek matice F_j taktéž hustý. Část zbývající k faktorizaci je označena jako U_j. U_j se nazývá update matice sloupce j (neplést si s subtree update maticí) a je taktéž hustá.

Věta 3 *Pro update matici platí*

$$U_j = - \sum_{k \in T[j]} \begin{pmatrix} l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{pmatrix} \begin{pmatrix} l_{i_1,k} & \dots & l_{i_r,k} \end{pmatrix}.$$

Je důležité rozlišovat mezi subtree update maticí $\overline{U}_j$ a update maticí U_j. Subtree update matice $\overline{U}_j$ je použita k vytvoření frontální matice F_j, zatímco update matice U_j je vypočítána z jednoho kroku samotné eliminace nad F_j. Update matice U_j má o jeden řádek a sloupec méně než subtree update matice $\overline{U}_j$. Tento „chybějící“ řádek už byl plně spočítán a proto je k dalším krokům nepotřebný. Struktura obou matic $\overline{U}_j$ a U_j je založena na sloupcové struktuře $L_{*,j}$. Zatímco U_j obsahuje součiny vnějších příspěvků všech sloupců podstromu $T[j]$, $\overline{U}_j$ obsahuje pouze příspěvky sloupců z $T[j] - \{j\}$.

Ukažme si nyní výpočet frontální, subtree update a update matice. Budeme vycházet z matice A na obrázku 2. Díky eliminačnímu stromu 3 můžeme ihned odvodit, že $\overline{U}_1 = 0, \overline{U}_2 = 0$ a $\overline{U}_3 = 0$. Jedná se totiž o listy, které nemají žádné potomky a proto jsou vnější součiny příspěvků nulové.

Spočtěme si nyní příslušné matice pro sloupec (uzel) číslo 4 [1]. Z eliminačního stromu vyčteme, že $T[4] - \{4\} = \{2\}$, subtree update matice a frontální matice tedy jsou

$$\overline{U}_4 = -\begin{pmatrix} l_{4,2} \\ l_{6,2} \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} l_{4,2} & l_{6,2} & 0 & 0 \end{pmatrix} = \begin{pmatrix} -l_{4,2}^2 & -l_{4,2}l_{6,2} & 0 & 0 \\ -l_{4,2}l_{6,2} & -l_{6,2}^2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

a proto

$$F_4 = -\begin{pmatrix} a_{4,4} & 0 & a_{4,8} & a_{4,9} \\ 0 & 0 & 0 & 0 \\ a_{4,8} & 0 & 0 & 0 \\ a_{4,9} & 0 & 0 & 0 \end{pmatrix} + \overline{U}_4 = \begin{pmatrix} a_{4,4} - l_{4,2}^2 & -l_{4,2}l_{6,2} & a_{4,8} & a_{4,9} \\ -l_{4,2}l_{6,2} & -l_{6,2}^2 & 0 & 0 \\ a_{4,8} & 0 & 0 & 0 \\ a_{4,9} & 0 & 0 & 0 \end{pmatrix}.$$

Jedním krokem eliminace nad frontální maticí F_4 spočteme faktor sloupce $L_{*,4}$ a hustou update matici U_4:

$$U_4 = \begin{pmatrix} -l_{6,2}^2 - l_{6,4}^2 & -l_{6,4}l_{8,4} & -l_{6,4}l_{9,4} \\ -l_{6,4}l_{8,4} & -l_{8,4}^2 & -l_{8,4}l_{9,4} \\ -l_{6,4}l_{9,4} & -l_{8,4}l_{9,4} & -l_{9,4}^2 \end{pmatrix}.$$

3.4 Vytváření frontálních a update matic

V předchozí části jsme probrali teoretický základ frontálních a update matic. Nyní si ukážeme, jak se tyto matice efektivně vytvářejí a jaký je mezi nimi vztah. Víme již, že frontální matice F_j je vytvořena pomocí subtree update matice $\overline{U}_j$. Následnou eliminací nad F_j pak vznikne update matice U_j, která se využívá v dalším kroku eliminace.

Pokud chceme formálně popsat vztah mezi frontálními a udpate maticemi, musíme nejdřív nadefinovat tzv. *extend-add* operaci. Mějme čtvercovou matici R, písmenem r označme její počet sloupců a řádků. Obecně platí $r \geq 1$, ale v našem případě se omezíme na $n \geq r \geq 1$ (n označuje velikost faktorizované matice A). Dále mějme čtvercovou matici S, písmenem s označme obdobně její velikost ($n \geq s \geq 1$). Každý řádek a sloupec matic R a S koresponduje s řádky a sloupci faktorizované matice A. Nechť $i_1 \leq \cdots \leq i_r$ je množina indexů řádků (a sloupců) matice A v matici R (první řádek matice R může např. reprezentovat třetí řádek v matici A). Stejným způsobem reprezentuje množina $j_1 \leq \cdots \leq j_s$ indexy v matici S.

Uvažujme nyní sjednocení těchto dvou množin. Nechť $k_1 \leq \cdots \leq k_t$ je výsledná množina tohoto sjednocení. Matice R může být rozšířena do tvaru, který vyhovuje výsledné množině indexů $\{k_1, \ldots, k_t\}$ přidáním nulových řádků a sloupců. Tyto řádky a sloupce reprezentují chybějící indexy, které jsou obsaženy v matici S. Stejným způsobem může být matice S rozšířena o chybějící indexy matice R. Provedeme-li operaci $R \oplus S$, dostaneme $t \times t$ matici T. Tato matice je vytvořena sečtením dvou rozšířených matic R a S. Maticový operátor „$\oplus$“ reprezentuje již

zmíněnou *extend-add* operaci. Toto zobecněné sčítání matic se nazývá maticová superpozice. Aplikaci *extend-add* operace si ukažme na příkladu. Mějme

$$R = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \text{ a } S = \begin{pmatrix} e & f \\ g & h \end{pmatrix},$$

kde množina $\{4, 7\}$ reprezentuje indexy řádků (a sloupců) obsažených v matici R a množina $\{4, 8\}$ reprezentuje indexy obsažené v matici S. Výsledkem operace $R \oplus S$ je následující 3×3 matice s množinou indexů $\{4, 7, 8\}$:

$$R \oplus S = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} e & 0 & f \\ 0 & 0 & 0 \\ g & 0 & h \end{pmatrix} = \begin{pmatrix} a+e & b & f \\ c & d & 0 \\ g & 0 & h \end{pmatrix}.$$

Vztah mezi frontální maticí F_j a množinou update matic U_j může být formálně popsán pomocí eliminačního stromu a *extend-add* operace. Uvažujme sloupce j, nechť $j, i_1, \ldots, i_r$ jsou řádkové indexy nenulových prvků ve sloupci $L_{*,j}$.

Věta 4 *Nechť uzly $c_1, \ldots, c_s$ jsou potomky uzlu j v eliminačním stromě. Potom platí*

$$F_j = \begin{pmatrix} a_{j,j} & a_{j,i_1} & \ldots & a_{j,i_r} \\ a_{i_1,j} & & & \\ \vdots & & 0 & \\ a_{i_r,j} & & & \end{pmatrix} \oplus U_{c1} \oplus \cdots \oplus U_{cs}.$$

Věta 4 je jedním ze základních pilířů multifrontální metody, proveďme si proto její důkaz.
Důkaz [1] Frontální matice F_j je v (3) definována pomocí $\overline{U}_j$. Jak můžeme vidět z (2), $\overline{U}_j$ vznikla shromážděním všech příspěvků vnějších součinů ze sloupců $T[j] - \{j\}$. Jelikož $c_1, \ldots, c_s$ jsou potomci uzlu j v eliminačním stromě, je $T[j] - \{j\}$ jednoduše disjunktní množina uzlů v podstromech $T[c_1], \ldots, T[c_s]$. Proto můžeme vyjádřit $\overline{U}_j$ jako seskupení vnějších součinů příspěvků ze sloupců v $T[c_1], \ldots, T[c_s]$. Podle 1 je pro každý strom $T[c_t]$ $(1 \leq t \leq s)$ jeho vnější součin příspěvků do F_j daný pomocí U_{c_t}. Proto jsou všechny příspěvky ze sloupců v $T[j] - \{j\}$ obsaženy v $U_{c_1}, \ldots, U_{c_s}$. Tímto je důkaz u konce. ∎

Submatice $U_{c_1} \oplus \cdots \oplus U_{c_s}$ může mít méně řádků a sloupců než frontální matice F_j. Pokud je ale správně rozšířena (tak, aby její množina indexů vyhovovala množině indexů matice F_j), stane se prakticky update maticí $\overline{U}_j$. Obsahuje tedy všechny vnější součiny příspěvků z $\overline{U}_j$ do frontální matice F_j. Duff a Reid popisují tento proces vytváření j-té frontální matice z $A_{*,j}$ a update matic její potomků jako *assembly* (sestavovací) operaci. Proto i eliminační strom často označují obecnějším názvem *assembly tree* (sestavovací strom).

Nyní známe všechny potřebné informace k formulaci Choleského řídké faktorizace pomocí frontálních a update matic. Algoritmus 1 zachycuje podstatu multifrontální metody, ale nezahr-

for *sloupec* $j := 1$ **to** n **do**

Nechť $j, i_1, \ldots, i_r$ jsou řádkové indexy nenulových prvků ve sloupci $L_{*,j}$;

Nechť $c_1, \ldots, c_s$ jsou potomci uzlu j v eliminačním stromě;

Vytvořme update matici $\overline{U} = U_{c_1} \oplus \cdots \oplus U_{c_s}$;

$$F_j = \begin{pmatrix} a_{j,j} & a_{j,i_1} & \ldots & a_{j,i_r} \\ a_{i_1,j} & & & \\ \vdots & & 0 & \\ a_{i_r,j} & & & \end{pmatrix} \oplus \overline{U};$$

Faktorizujme F_j do

$$\begin{pmatrix} l_{j,j} & 0 \\ l_{i_1,j} & \\ \vdots & I \\ l_{i_r,j} & \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & U_j \end{pmatrix} \begin{pmatrix} l_{j,j} & l_{i_1,j} \ldots l_{i_r,j} \\ 0 & I \end{pmatrix};$$

end

Algoritmus 1: Choleského faktorizace pomocí frontálních a update matic

nuje vyřešení některých problémů, které nejsou na první pohled zřejmé. Tyto problémy budou detailně vysvětleny v kapitole o implementaci.

3.5 Grafová interpretace

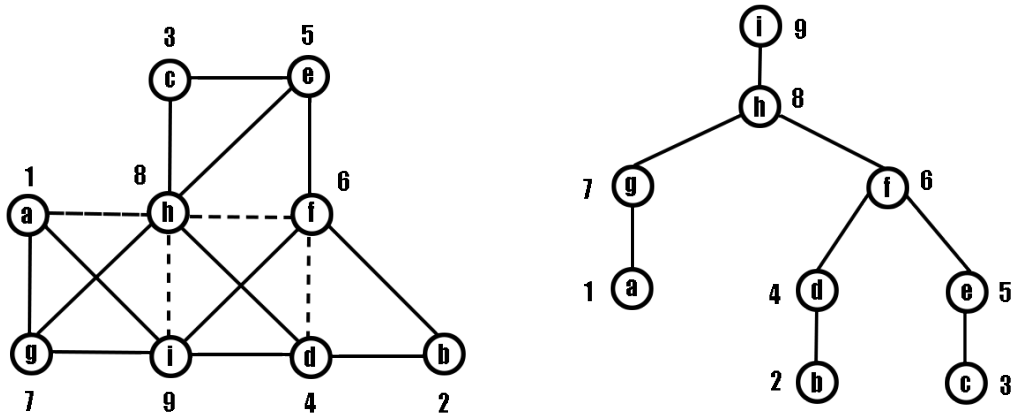
V této části si přiblížíme grafovou teorii stojící za frontálními a update maticemi. Tato část také slouží jako prerekvizita k supernodální verzi multifrontální metody.

Nechť je G neorientovaný graf reprezentující danou řídkou $n \times n$ matici A. Bez ztráty na obecnosti předpokládejme, že je graf G souvislý. Nyní uvažujme souvislou podmnožinu uzlů C v daném grafu. Fronta podmnožiny C je definována jako množina vedlejších uzlů v grafu G. Pro zapsání této fronty použijeme označení $Adj_G(C)$.

Podstrom $T[j]$ s kořenem j v eliminačním stromě udává souvislý podgraf v grafu G [1]. Řádkové a sloupcové indexy frontální matice F_j lze vyjádřit jako $\{j\} \cup Adj_G(T[j])$. Indexy update matice U_j lze vyjádřit jako $Adj_G(T[j])$.

Zaměřme se nyní na graf z obrázku 4. Plné čáry reprezentují nenulové prvky v matici A, zatímco přerušované čáry reprezentují doplňky v Choleského faktoru L. Pro uzel číslo 4 je $Adj_G(T[4]) = Adj_G(\{4, 2\}) = \{6, 8, 9\}$, což je množina indexů nenulových prvků v update matici U_4. Pro uzel číslo 6 platí $Adj_G(T[6]) = Adj_G(\{2, 3, 4, 5, 6\}) = \{8, 9\}$, což je množina indexů nenulových prvků v update matici U_6.

Pokud se na eliminační proces díváme z grafového hlediska, vidíme, že samotný proces probíhá na několika frontách. Po eliminaci uzlu číslo 5 existují tři podstromy $T[1] = \{1\}$, $T[4] = \{2, 4\}$ a $T[5] = \{3, 5\}$, které mají eliminované uzly. Tyto podstromy odpovídají třem souvislým podgrafům ve výchozím grafu. Jejich množiny vedlejších uzlů jsou tři fronty, ve kterých



Obrázek 4: Graf a eliminační strom matice z obrázku 2

proběhla eliminace. Tyto fronty jsou dány jako $Adj_G(T[1]) = \{7, 8, 9\}$, $Adj_G(T[4]) = \{6, 8, 9\}$ a $Adj_G(T[5]) = \{6, 8\}$.

Uvažujme nyní vytvoření frontální matice F_6 pomocí matice $\overline{U}_6$. Díky věty 4 víme, že $\overline{U}_6 = U_4 \oplus U_5$. Tento krok si můžeme vyložit jako sloučení dvou front $Adj_G(T[4])$ a $Adj_G(T[5])$. Sjednocením těchto dvou front vznikne množina $\{6, 8, 9\}$. Po eliminaci uzlu číslo 6 vznikne nová fronta $\{8, 9\}$, která reprezentuje $Adj_G(T[6])$.

Jeden krok eliminace nad frontální matici F_j sdružený s uzlem j v eliminačním stromě vede k vytvoření nové fronty. Tato fronta může být vytvořena dvěma způsoby. Pokud je j list, nová fronta $Adj_G(T[j])$ bude přidána do množiny front. Pokud j není list, nová fronta vznikne sloučením front všech potomků uzlu j.

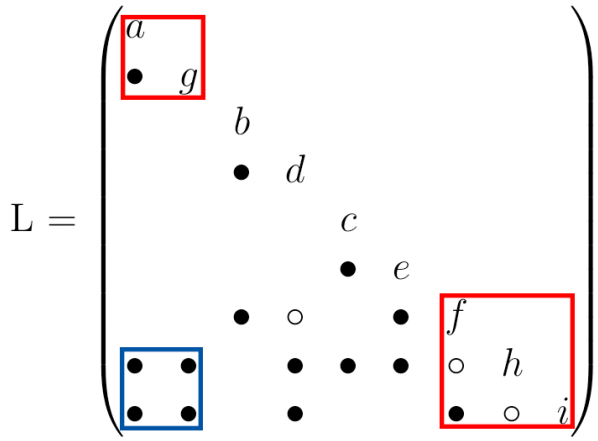
3.6 Supernodální verze

V této části si představíme praktické vylepšení multifrontální metody - tzv. supernody. Jak jsme si již ukázali v předchozích sekcích, jedním krokem eliminace vypočítáme jeden sloupec výsledného faktoru L. Jinými slovy musíme provést n kroků eliminace pro faktorizaci celé $n \times n$ matice A. Představme si nyní, že řádky a sloupce matice A nějakým způsobem seskupíme a s tímto seskupením budeme pracovat jako s jednou výpočetní jednotkou. Provedením eliminace nad touto jednotkou vypočteme najednou t sloupců výsledného faktoru L, což samozřejmě urychlí celý proces faktorizace.

Definujme nyní formálně toto seskupení ve smyslu eliminačního stromu. Supernode je maximální množina souvislých uzlů $\{j, j+1, \ldots, j+t\}$, pro které platí

$$Adj_G(T[j]) = \{j, j+1, \ldots, j+t\} \cup Adj_G(T[j+t]). \tag{4}$$

Z této definice vyplývá, že pokud v supernodu $\{j, j+1, \ldots, j+t\}$ uvažujeme uzel $j+k$ $(1 \leq k \leq t)$, je uzel $j+k-1$ jeho potomkem v eliminačním stromě.



Obrázek 5: Ukázka dvou supernodů. Červená oblast označuje diagonální blok, modrá totožnou strukturu sloupců

Z maticového pohledu udává $Adj_G(T[j])$ množinu indexů nenulových prvků ve sloupci $L_{*,j}$. Supernode odpovídá maximálnímu bloku sloupců v Choleského faktoru L, který je trojúhelníkového tvaru a jeho sloupce mají mimo daný blok totožnou strukturu (obrázek 6).

Uvažujme nyní množinu supernodů a supernodální eliminační strom. Supernode S je rodičem supernodu $T = \{j, \ldots, j+t\}$, pokud rodič s uzlu $j+t$ v klasickém eliminačním stromě náleží do S.

Zavedení supernodu má samozřejmě vliv na vytváření frontálních matic. Definujme nyní supernodální frontální matici. Mějme supernode $S = \{j, \ldots, j+t\}$, $j+t, i_1, \ldots, i_r$ jsou řádkové indexy nenulových prvků ve sloupci $T_{*,j+t}$ (poslední sloupec supernodu S). Supernodální frontální matice odpovídající S vypadá následovně:

$$\mathcal{F}_j = \begin{pmatrix} a_{j,j} & a_{j,j+1} & \ldots & a_{j,j+t} & a_{j,i_1} & \ldots & a_{j,i_r} \\ a_{j+1,j} & a_{j+1,,j+1} & \ldots & a_{j+1,j+t} & a_{j+1,,i_1} & \ldots & a_{j+1,i_r} \\ \vdots & \vdots & \ldots & \vdots & & & \\ a_{j+t,j} & a_{j+t,j+1} & \ldots & a_{j+t,j+t} & a_{j+t,i_1} & \ldots & a_{j+t,i_r} \\ a_{i_1,j} & a_{i_1,j+1} & \ldots & a_{i_1,j+t} & & & \\ \vdots & \vdots & \ldots & \vdots & & 0 & \\ a_{i_r,j} & a_{i_r,j+1} & \ldots & a_{i_r,j+t} & & & \end{pmatrix} + \overline{\mathcal{U}}_j,$$

kde

$$\overline{\mathcal{U}}_j = - \sum_{\substack{k \in T[j+t] \\ -\{j,\ldots,j+t\}}} \begin{pmatrix} l_{j,k} \\ l_{j+1,k} \\ \vdots \\ l_{j+t,k} \\ l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{pmatrix} \begin{pmatrix} l_{j,k} & l_{j+1,k} & \ldots & l_{j+t,k} & l_{i_1,k} & \ldots & l_{i_r,k} \end{pmatrix}.$$

Pomocí supernodální frontální matice $\mathcal{F}_j$ a update matic můžeme sestavit supernodální verzi věty 4. Při vytváření matice $\mathcal{F}_j$ pomocí *extend-add* operace zahrnujeme všechny update matice U_i. Tyto matice představují update matice „nejvyššího“ uzlu každého supernodálního potomka - supernode $S = \{j, j+1, \ldots, j+t\}$ je složen z několika uzlů, update matice U_i tedy představuje update matici uzlu $j + t$. Je důležité uvědomit si, že pro supernode $S = \{j, j + 1, \ldots, j + t\}$ mají jeho frontální matice F_j i supernodální frontální matice $\mathcal{F}_j$ stejnou velikost a stejnou množinu původních indexů. Po formulaci matice $\mathcal{F}_j$ bylo jeho prvních $t + 1$ sloupců kompletně aktualizováno (získaly veškeré příspěvky vnějších součinů z předchozích sloupců) a proto tyto sloupce mohou být eliminovány najednou.

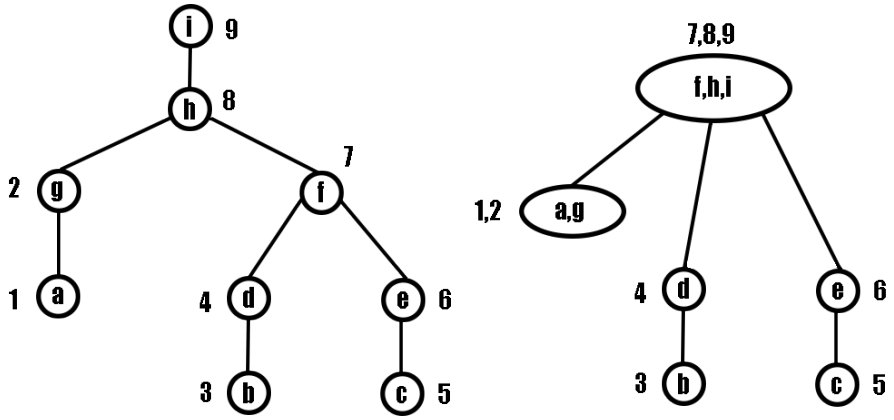
Nyní můžeme zformulovat supernodální verzi multifrontální metody popsané v algoritmu 1. I v této formulaci se předpokládá, že jednotlivé supernody jsou seřazeny ve stejném pořadí jako sloupce vstupní matice A.

for *každý supernode S ve vzestupném pořadí* **do**
  Nechť $S = \{j, \ldots, j + t\}$;
  Nechť $j + t, i_1, \ldots, i_r$ jsou řádkové indexy nenulových prvků ve sloupci $L_{*,j+t}$;

$$\mathcal{F}_j = \begin{pmatrix} a_{j,j} & a_{j,j+1} & \ldots & a_{j,j+t} & a_{j,i_1} & \ldots & a_{j,i_r} \\ a_{j+1,j} & a_{j+1,,j+1} & \ldots & a_{j+1,j+t} & a_{j+1,,i_1} & \ldots & a_{j+1,i_r} \\ \vdots & \vdots & \ldots & \vdots & & & \\ a_{j+t,j} & a_{j+t,j+1} & \ldots & a_{j+t,j+t} & a_{j+t,i_1} & \ldots & a_{j+t,i_r} \\ a_{i_1,j} & a_{i_1,j+1} & \ldots & a_{i_1,j+t} & & & \\ \vdots & \vdots & \ldots & \vdots & & 0 & \\ a_{i_r,j} & a_{i_r,j+1} & \ldots & a_{i_r,j+t} & & & \end{pmatrix};$$

  $n :=$ počet potomků supernodu S v eliminačním stromě;
  for $i := 1$ **to** n **do**
    Vytvořme $\mathcal{F}_j = \mathcal{F}_j \oplus U$;
  end
  Vykonejme $t + 1$ kroků eliminace na frontální matici $\mathcal{F}_j$ k získání sloupců $L_{*,j}, \ldots, L_{*,j+t}$ a update matice U_{j+t};
end

Algoritmus 2: Choleského faktorizace pomocí supernodální multifrontální metody



Obrázek 6: Porovnání klasického eliminačního stromu (nalevo) s jeho supernodální verzí

3.7 LU rozklad

Na začátku této kapitoly jsme zmínili, že multifrontální metodu lze aplikovat na symetrické i nesymetrické matice, záleží jaký typ faktorizace uvnitř multifrontální metody použijeme. Pokud pracujeme s Choleského faktorizací, jsme omezeni pouze na symetrické matice. Při použití LU rozkladu máme volnější ruku a můžeme faktorizovat jak symetrické, tak nesymetrické matice. V této části si povrchově představíme multifrontální metodu s použitím LU rozkladu, princip multifrontální metody ovšem zůstává stejný, takže nepůjdeme příliš do detailu.

Uveďme si nyní základní popis LU rozkladu. Mějme čtvercovou matici A, její faktorizace do tvaru $A = LU$ existuje a je jednoznačná, pokud všechny hlavní minory matice A jsou nenulové. Matice L je potom dolního trojúhelníkového tvaru a matice U horního trojúhelníkového tvaru. Jeden krok faktorizace husté $n \times n$ matice A pomocí LU rozkladu můžeme zapsat jako

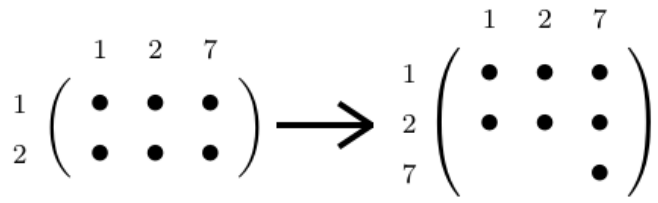
$$A = \begin{pmatrix} b & t \\ v & C \end{pmatrix} = \begin{pmatrix} I & 0 \\ v/b & I \end{pmatrix} \begin{pmatrix} b & 0 \\ 0 & C - vt/b \end{pmatrix} \begin{pmatrix} I & t/b \\ 0 & I \end{pmatrix},$$

kde b je první diagonální element, v a t jsou vektory o velikost $(n-1)$ a submatice $C - vt/b$ je část zbývají k faktorizaci. Po vykonání i kroků faktorizace $(i > 1)$ je výhodné zapsat tuto částečnou faktorizaci pomocí blokového rozdělení:

$$A = \begin{pmatrix} B & T \\ V & C \end{pmatrix} = \begin{pmatrix} I & 0 \\ VB^{-1} & I \end{pmatrix} \begin{pmatrix} B & 0 \\ 0 & C - VB^{-1}T \end{pmatrix} \begin{pmatrix} I & B^{-1}T \\ 0 & I \end{pmatrix}.$$

Základní myšlenka řídké LU faktorizace zůstala stejná - pomocí eliminačního stromu a frontálních matic postupně faktorizujeme všechny sloupce matice A. Pokud je zadaná matice A symetrická, z LU rozkladu se stává Choleského faktorizace. Pokud je matice A nesymetrická, situace se poněkud zkomplikuje.

U nesymetrické matice vyvstávají dva základní problémy - tvorba eliminačního stromu a práce s frontálními maticemi. Duff a Reid vyvinuli postup pro vytvoření správného elimi-



Obrázek 7: Ukázka „extend-add“ operace na samostatné matici

načního stromu. Uvažujme matici $M = A + A^T$ (samotný součet je proveden symbolicky). Eliminační strom nad symetrickou maticí M je pak použit k LU rozkladu nesymetrické matice A. Jinými slovy můžeme říct, že k LU rozkladu nesymetrická matice A je použit eliminační strom Choleského faktorizace symetrické matice M [4].

Procesem sloučení (amalgamace) lze eliminační strom transformovat do jeho „supernodální“ podoby. Amalgamované uzly jsou vytvořeny na základě podobné podmínky jako supernody - pokud má rodič a jeho jediný potomek stejnou strukturu nenulových prvků, můžeme z těchto uzlů vytvořit amalgamovaný uzel. Všimněme si, že tento uzel nemůžeme nazvat supernodem, jelikož pro jeho vytvoření musíme splnit přísnější kritérium - rodič musí mít pouze jediného potomka.

Díky nesymetrické struktuře matice A mohou být i jednotlivé frontální matice nesymetrické. Pokud bychom k vytváření těchto matic přistupovali stejným způsobem jako u Choleského faktorizace, vznikaly by obecně frontální matice obdélníkového tvaru. Jedním řešením je přistupovat k řádkovým a sloupcovým indexům odděleně a matici rozšířit „samu nad sebou“ (obrázek 7). Tímto způsobem vznikne čtvercová frontální matice, na kterou již lze aplikovat jeden či více kroků eliminace. Kromě komplikací s eliminačním stromem a frontálními maticemi, LU verze multifrontální metody již neskrývá žádná další překvapení.

4 Vlastní implementace multifrontální metody

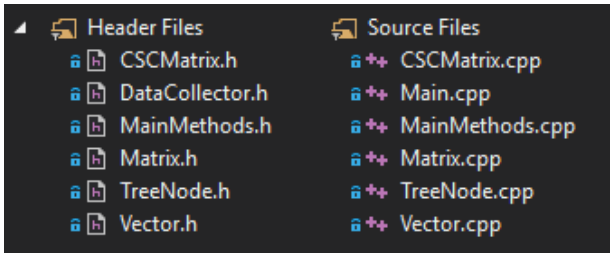
V předchozí kapitole jsme probrali vše potřebné ke správnému pochopení multifrontální metody, nyní si popíšeme její praktickou implementaci. Tato část bude obsahovat kompletní popis algoritmu, diskuzi nejvýznamnějších problémů a vysvětlení některých programátorských praktik.

Celá implementace byla vytvořena objektově orientovaným způsobem s důrazem na kvalitní a přehledný kód. Program byl napsán v jazyce C++ ve standardu C++11 a nevyužívá žádné nadstandardní knihovny. Program není předmětem autorských práv a je volně dostupný na GIT repozitáři `https://gitlab.com/Mihula/diploma`.

4.1 Struktura programu

Seznamme se nejdříve se samotnou strukturou programu (obrázek 8). Hlavičkové soubory jsou použity převážně k deklaraci tříd a obsahují pouze základní metody - konstruktor, destruktor, gettery a settery. Výjimkou je hlavičkový soubor *MainMethods.h*, ten obsahuje prototypy funkcí, které jsou následně implementovány ve zdrojovém souboru *MainMethods.cpp*. Tyto funkce svou povahou nesouvisí s žádnou třídou a proto jsou tímto způsobem odděleny. *Main.cpp* obsahuje hlavní funkci programu, která má na starost celkový běh programu a volá všechny ostatní funkce. Složitější metody jednotlivých tříd jsou implementovány ve zdrojových souborech se stejným názvem. Popišme si nyní stručně jednotlivé třídy:

- *CSCMatrix* reprezentuje vstupní matici v řídkém formátu
- *DataCollector* je třída určena k zaznamenání časů a dalších informací - jako jediná třída nemá díky své jednoduchosti příslušný zdrojový soubor
- *TreeNode* je nejdůležitější třída programu, reprezentuje jeden uzel eliminačního stromu a nese veškerou logiku multifrontální metody, detailně si tuto třídu popíšeme v další části
- *Matrix* a *Vector* jsou obalové třídy pro husté matice a vektory. Tyto třídy navíc obsahují množinu svých původních indexů (pro *extend-add* operaci)



Obrázek 8: Struktura programu

4.2 Formáty řídkých matic a načtení dat

Řídké matice jsou z převážné většiny tvořeny nulovými prvky. Je neefektivní tyto prvky ukládat nebo s nimi jakýmkoliv způsobem pracovat. Z toho důvodu byly vytvořeny formáty řídkých matic, které umožňují efektivnější uložení do paměti a provádění operací (klasické maticové operace, faktorizace, apod.).

Formát COO

COO (Coordinate) neboli souřadnicový formát je z pohledu implementace velice jednoduchý. Mějme $n \times n$ matici A mající k ($k > n$) nenulových prvků $a_{i,j}$. Souřadnicová reprezentace je uspořádaná trojice $(i, j, a_{i,j})$. Pro uložení celé matice A je potřeba $3k$ čísel, platí-li $3k < n^2$, pak je souřadnicový formát úspornější. Z pohledu provádění maticových operací není souřadnicový formát vhodný a proto se nejčastěji používá k přenosu dat mezi jednotlivými stanicemi.

Formát CSC a CSR

CSC (Compressed Sparse Columns) je standardní formát pro ukládaní řídkých matic. Řídká matice je uložena do tří polí. První pole obsahuje všechny nenulové prvky seřazené po sloupcích zleva doprava. Druhé pole obsahuje řádkové indexy těchto prvků. Třetí pole obsahuje „ukazatele“ na začátky jednotlivých sloupců (z pohledu celkového počtu prvků do daného sloupce). Pro vysvětlení uvažujme matici

$$A = \begin{pmatrix} 3 & & 6 & \\ & 7 & & 8 \\ & 4 & 5 & \\ & 1 & & 1 \end{pmatrix},$$

reprezentace v CSC formátu pak vypadá následovně:

hodnoty $= [3, 7, 4, 1, 6, 5, 8, 1]$

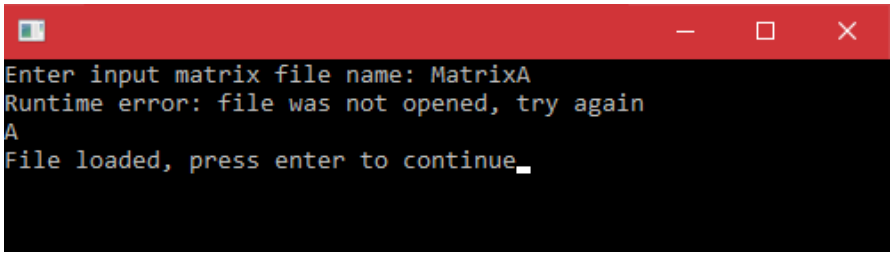
řádkové indexy $= [1, 2, 3, 4, 1, 3, 2, 4]$

ukazatele $= [0, 1, 4, 6, 8]$

Pro uložení $n \times n$ matice A mající k nenulových prvků je potřeba $2k + n$ čísel. Pokud místo sloupců chceme pracovat s řádky, použijeme analogicky formát CSR (Compressed Sparse Rows).

Převod z COO do CSC

Program očekává na vstupu matici v seřazeném souřadnicovém formátu. Řídká matice může být v souřadnicovém formátu zapsaná „chaoticky“, tzn. jednotlivé trojice nejsou seřazeny podle řádkových a následně sloupcových indexů. Tento zápis stále korektně uchovává danou matici, ale zbytečně komplikuje převod do jiného formátu. Samotný algoritmus



Obrázek 9: Ukázka načtení matice ze souboru s názvem „A“

převodu (předpokládající seřazené trojice) je poměrně jednoduchý, postupně se načítají trojice a kontroluje se, zda došlo ke změně sloupcového indexu, pokud ano, zapíše se ukazatel na nový sloupec a pokračuje se dále.

Reprezentace v programu

Díky rychlému přístupu ke sloupcům (u symetrické matice i řádkům) je formát CSC nejvhodnější k reprezentaci vstupní matice. Třída *CSCMatrix* v sobě uchovává potřebné informace.

```
class CSCMatrix {
private:
   std::vector<double> values = std::vector<double>();
   std::vector<int> rows = std::vector<int>();
   std::vector<int> columns = std::vector<int>();
public:
   CSCMatrix() {}
};
```

Výpis 1: Ukázka třídy *CSCMatrix*

Prvním krokem programu je načtení vstupní matice A. Program načítá data z textového souboru. Pokud soubor s maticí neexistuje nebo není korektně načten, uživatel má možnost operaci opakovat.

4.3 Vytvoření eliminačního stromu

První krokem samotné multifrontální metody je vytvoření eliminačního stromu. Existuje více možností jak v programu eliminační strom (obecně jakýkoliv strom) uchovávat. Nejčastěji se setkáme s uložením pomocí klasického pole, tento způsob je efektivní, ale takto vytvořený strom je poměrně nepřehledný. V programu uchováváme strom pomocí třídy *TreeNode*. Každý uzel stromu je reprezentován jednou instancí dané třídy. Třída obsahuje ukazatele na své potomky a rodiče, takže se ve stromě můžeme snadno pohybovat oběma směry, což je vlastnost, kterou budeme potřebovat. Popišme si nyní detailně strukturu třídy *TreeNode*.

```
class TreeNode {
private:
   std::vector<int> indeces = std::vector<int>();
   std::vector<TreeNode*> children = std::vector<TreeNode*>();
   TreeNode* parent = NULL;
   std::set<int>* adjacents = new std::set<int>();
   Matrix* updateMatrix = NULL;
   bool ready = false;
public:
   TreeNode() {}
   TreeNode(int index) { indeces.push_back(index); }
};
```

Výpis 2: Ukázka třídy *TreeNode*

Každý uzel eliminačního stromu reprezentuje alespoň jeden sloupec matice A (supernode reprezentuje více sloupců). Při vytváření frontální matice je potřeba znát indexy sloupců, které daný uzel reprezentuje. Tyto indexy jsou uloženy ve vektoru s názvem *indeces*. Veškeré instance třídy *TreeNode* jsou vytvořeny pouze jednou a v programu se předávají pomocí ukazatelů. Vektor ukazatelů *children* obsahuje ukazatele na své potomky. Ukazatel *parent* zase ukazuje na rodiče daného uzlu. Při transformaci eliminačního stromu na jeho supernodální verzi je třeba znát množinu vedlejších uzlů (jak bylo zmíněno v sekci 3.5), tato množina je dočasně uložena v setu *adjacents*. Boolean proměnná *ready* popisuje, zda už nad daným uzlem proběhl proces eliminace. Datovou strukturu eliminačního stromu máme definovanou, popišme si nyní samotný

for *sloupec* $j := 0$ **to** $n - 1$ **do**
  parent(j):= ∞;
  for *index* $i := 0$ **to** $|R_j| - 1$ **do**
    $r = c_i$;
    while *parent(r)* $\neq \infty$ **do**
      r:=parent(r);
    end
    if $r \neq j$ **then**
      parent(r) = j;
    end
  end
end

Algoritmus 3: Vytvoření eliminačního stromu [2]

proces vytvoření. Na vstupu mějme matici A, označme $R_j = \{c_o, \ldots, c_{|R_j|-1}\}$ jako množinu nenulových prvků i v řádku j pro které platí $i < j$. Algoritmus 4 udává proces vytvoření eliminačního stromu. Jelikož máme matici uloženou v CSC formátu, je přístup k řádkům velice neefektivní. Naštěstí můžeme využít symetrie matice a při získávání množiny nenulových prvků

R_j načteme příslušný sloupec. Operace získání daného sloupce z formátu CSC má lineární složitost (závisí pouze na počtu prvků v daném sloupci) a je proto efektivní. Po dokončení algoritmu získáme vektor navzájem propojených uzlů představující klasický eliminační strom. Vektor je použit pouze jako jednoduchý způsob uchování stromu „na jednom místě“.

4.3.1 Transformace eliminačního stromu

Dalším krokem je transformace eliminačního stromu do jeho supernodální podoby. Jednotlivé uzly jsou ve výstupním vektoru seřazeny vzestupně jako indexy sloupců v matici A. Jelikož je supernode seskupení souvislých uzlů (sloupců) splňujících (4), vlastnost vzestupného seřazení nám dává jednoduchý způsob transformace do supernodální podoby. Transformace uzlu na supernode v 4 představuje přidání indexů a převedení potomků z „pohlcených“ uzlů do supernodu a nastavení správného rodiče. Rodič je získán z posledního uzlu daného supernodu. První uzel je tedy transformován na supernodální, ostatní uzly jsou pohlceny a jejich instance zahozeny.

for *uzel* $i := 0$ **to** $n - 1$ **do**
  for *uzel* $j := i + 1$ **to** $n - 1$ **do**
    if $Adj_G(T[i]) = \{i, \ldots, j\} \cup Adj_G(T[j])$ **then**
      transformujme uzel i na supernode a přidejme do něj uzel j;
    else
      vyskočme z vnitřního cyklu a inkrementujme i o velikost supernodu j;
    end
  end
end

Algoritmus 4: Transformace eliminačního stromu do supernodální podoby

4.4 Extend-add operace

Prvním optimalizačním problémem programu byla efektivní implementace *extend-add* operace. Uvažujme proces vytvoření nějaké frontální matice F:

$$F = \begin{pmatrix} \bullet & \bullet & \bullet \\ \bullet & & \\ \bullet & & \end{pmatrix} \oplus \begin{pmatrix} \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \end{pmatrix} \oplus \begin{pmatrix} \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \end{pmatrix},$$

kde množiny indexů nenulových prvků daných matic jsou $\{1, 4, 5\}$, $\{2, 4, 6\}$ a $\{2, 5, 6\}$. Množina indexů nenulových prvků výsledné frontální matice vznikne sjednocením všech vstupních množin, tedy $\{1, 2, 4, 5, 6\}$. Nyní je třeba jednotlivé matice rozšířit a sečíst. První možností je iterovat nad výslednou množinou indexů a porovnávat jednotlivé prvky s prvky vstupních množin. Tento přístup není vyloženě špatný, ale jeho implementace bude obsahovat spoustu zbytečných podmínek a iterací. Lepším způsobem je využití tzv. dvojité indexace. Každý z původních indexů bude mít ve výsledné matici nové místo. Vezměme původní množinu $\{2, 4, 6\}$ a porovnejme ji

s výslednou množinou $\{1, 2, 4, 5, 6\}$. Zatímco v původní množině byl index 2 na prvním místě, ve výsledné množině je až na třetím. Pokud budeme mít „tabulku“ původních a nových indexů, můžeme jednoduše projít každou matici právě jednou a přičíst dané prvky na správnou pozici do výsledné matice F. Danou implementaci můžeme popsat v následujících krocích:

1. Vytvořme výslednou množinu indexů nenulových prvků sjednocením všech vstupních množin
2. Pomocí mapy vytvořme „tabulku“ pro dvojitou indexaci
3. Vytvořme nulovou hustou matici F (její rozměry jsou stejné jako velikost výsledné množiny indexů)
4. Pro každou vstupní matici projděme její prvky a pomocí dvojité indexace tyto prvky přičtěme na správný index do výsledné matice F

4.5 Práce s pamětí

Správná práce s pamětí je alfou a omegou každého komplexnějšího programu. Přestože jazyk C++ poskytuje třídy a dává programátorovi možnost postavit vlastní vrstvu abstrakce a tudíž tak zvýšit úroveň pojetí, se kterou programátor pracuje, C++ stále patří mezi nízkoúrovňové jazyky. Jednou z výhod (z určitého pohledu i nevýhod) nízkoúrovňového jazyka je plná kontrola nad prací s pamětí. Programátoři přecházející z vysokoúrovňového jazyka (např. Java) na C++ mají často problém tento koncept pochopit. Vysokoúrovňové jazyky mají k dispozici tzv. garbage collector, který sám kontroluje paměť a uvolňuje bloky, na které již neexistuje žádná reference. Správná práce s pamětí má obrovský vliv na rychlost programu a přehlednost kódu a proto si v této sekci popíšeme základní rozdíly.

4.5.1 Heap versus stack

Heap je segment paměti určený k dynamické alokaci. Veškeré objekty uloženy na heapu musí být manuálně uvolněny. Pokud daný kus paměti není uvolněn, může dojít k tzv. memory leaku. V C++ se dynamická paměť alokuje pomocí operátoru *new*. Takto alokovaný blok musí být uvolněn pomocí operátoru *delete* dokud na něj máme referenci. Pokud přijdeme o referenci dříve, než paměť uvolníme - dostaneme se mimo scope (rámec), změníme danou referenci, apod., zůstane blok v paměti viset. Hlavní výhody a nevýhody heapu jsou:

- Alokace paměti na heapu je poměrně pomalá
- Alokovaná paměť zůstane alokovaná, dokud ji manuálně neuvolníme nebo program neskončí (po skončení programu by se měl o vyčištění paměti postarat operační systém)
- Heap má k dispozici velkou část paměti a je proto vhodný k udržení velkých datových struktur

Stack segment je část paměti, určená ke statické alokaci. Data uložená na stacku jsou alokovány i uvolněny automaticky, což pro programátora znamená méně práce se správou paměti a lépe čitelný kód. Stack segment je přístupný pomocí stack registru procesoru a funguje na principu LIFO - last in, first out. Alokace paměti na stacku je implementována pomocí inkrementace/dekrementace ukazatele na stack registru. Hlavní výhody a nevýhody stacku jsou:

- Alokace paměti na stacku je rychlá
- Alokovaná paměť zůstane alokovaná dokud je přítomna v aktuálním scopu, po opuštění scopu je vyjmuta ze stacku a automaticky uvolněna
- Stack je relativně malý a proto není vhodný k udržení velkých datových struktur

```
int functionOne() {
   TreeNode* heapNode = new TreeNode(); // dynamicky alokovana pamet (heap)
   TreeNode stackNode = TreeNode(); // automaticky alokovana pamet (stack)
   delete heapNode; // uvolneni dynamicky alokovane pameti
   return 1; // opoustime scope, stackNode je uvolnen automaticky
}
int functionTwo() {
   TreeNode* heapNode = new TreeNode();
   heapNode = new TreeNode(); // memory leak, predchozi instance nebyla
       uvolnena a zustava v pameti
   return 2; // prichazime o referenci i na druhou instanci, nastava dalsi
       memory leak
}
```

Výpis 3: Ukázka dynamické a automatické alokace paměti

Nepsaným pravidlem C++ je vyhýbat se dynamické alokaci paměti, pokud to není nutné nebo výhodné. V programu si dynamicky uchováváme instance třídy *TreeNode*, jelikož je struktura eliminačního stromu poměrně veliká a dostupná po celou dobu programu, je tento přístup bezpečný a vhodný. Třídní proměnné třídy *TreeNode*, které jsou alokovány automaticky jsou taktéž vytvořený na heapu, jelikož se automatická alokace pamětí řídí podle scopu ve kterém se momentálně nachází.

Dalším důležitým programátorským přístupem je předávání parametrů pomocí reference. Pokud do funkce pošleme parametr pomocí hodnoty, vytvoří se ve scopu dané funkce její kopie se kterou se v celé funkci pracuje. První nevýhodou tohoto přístupu je vytváření kopií vstupních parametrů při každém zavolání funkce - pokud je jedním ze vstupních parametrů nějaký „větší“ objekt, je tento proces pomalý a má vyšší paměťovou náročnost. Druhou nevýhodou je nemožnost upravit vstupní parametr v dané funkci - upravujeme pouze kopii, která je po průběhu

funkce zahozena. Pokud předáme vstupní parametr pomocí reference, vyhneme se oběma těmto problémům.

```
void function(int& a) { // predani vstupniho parametru pomoci reference
   a = 5;
}
int main() {
   int number = 7;
   std::cout << number << std::endl; // vytiskne cislo 7
   function(number);
   std::cout << number << std::endl; // vytiskne cislo 5
   return 1;
}
```

Výpis 4: Předávání parametrů pomocí reference

4.6 Popis algoritmu

Popišme si nyní celý proces vyřešení zadaného systému lineárních rovnic. První tři kroky v 4.6 jsme si již osvětlili, zaměřme se tedy na samotný proces faktorizace. Z teoretické části víme, že eliminace nad daným uzlem může proběhnout pouze tehdy, pokud již byli eliminování všichni jeho potomci. Z tohoto faktu vyplývá, že eliminace vždy začíná v jednotlivých listech stromu a postupně probublává až ke kořenům - jak již bylo zmíněno v teoretické části, náš eliminační strom může být ve skutečnosti les (skupina stromů), takže i kořenů může být více. Třída *TreeNode* obsahuje metodu s názvem *performElimination*, jejíž vstupní parametry jsou vstupní matice (nutná při formování frontální matice), výstupní matice (slouží k zápisu výsledného faktoru, v každém kroku je do ní vložen jeden či více eliminovaných sloupců) a vektor pro sběr kořenů - s kořeny budeme v dalším kroku pracovat a jelikož tímto způsobem procházíme všechny uzly, můžeme si kořeny již připravit. Po zavolání metody *performElimination* nad daným uzlem nejdříve proběhne kontrola, zda můžeme vůbec eliminaci vykonat. Při kontrole zjišťujeme, zda byl nad všemi potomky vykonán proces eliminace a jejich update matice jsou připraveny k použití. Pokud ano, vytvoříme frontální matici aktuálního uzlu a spočteme eliminovaný sloupec, který následně vložíme do výsledné matice. Pokud aktuální uzel není kořen a má tedy smysl počítat jeho update matici, výpočet provedeme. Po výpočtu update matice nastavíme boolean proměnnou *ready* na *true* a rekurzivně provoláme stejnou metodu nad rodičem aktuálního uzlu. Pokud je aktuální uzel kořen, přidáme ho do již zmiňovaného vektoru kořenů. Tímto způsobem postupně eliminujeme všechny sloupce a vypočteme výsledný faktor.

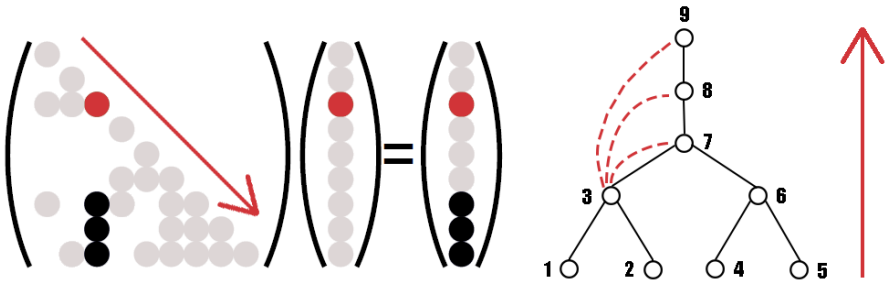
Po získání výsledného faktoru program přejde do fáze řešení. Tato fáze bude spolu s popisem výstupu programu detailněji popsána v následujících částech.

1. Načtení vstupních dat do formátu CSC

2. Vytvoření základního eliminačního stromu
3. Transformace eliminačního stromu do jeho supernodální podoby, zjištění všech jeho listů
4. Paralelní spuštění procesu eliminace nad všemi listy, zjištění všech kořenů
5. Načtení pravé strany řešené soustavy
6. Paralelní spuštění procesu dopředné substituce nad všemi listy,
7. Paralelní spuštění procesu zpětné substituce nad všemi kořeny
8. Výstup výsledků
9. Výstup naměřených časů a dalších informací

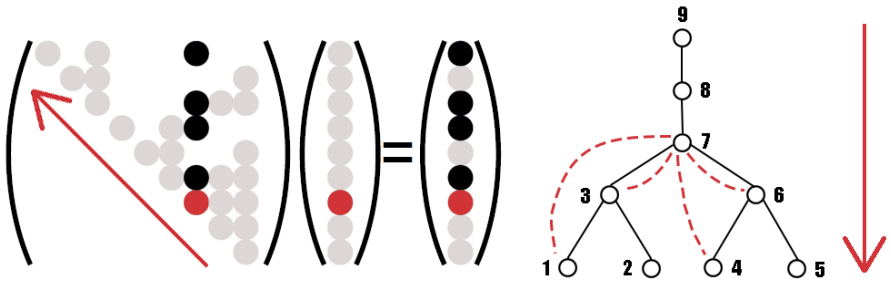
4.7 Dopředná a zpětná substituce

Jakmile získáme výsledný faktor, program přechází do fáze řešení. Nejdříve je z textového souboru načten vektor pravé strany. Soubor má uložen jednotlivé hodnoty ve sloupcovém formátu. Jak bylo zmíněno v 2.2, fáze řešení se skládá ze dvou částí - dopředné a zpětné substituce. I ve fázi řešení využíváme eliminační strom (obrázek 10). Třída *TreeNode* obsahuje metodu *performForwardSubstitution*, která je spuštěna na listech eliminačního stromu a rekurzivním způsobem postupuje směrem ke kořeni (obdobně jako u samotné faktorizace). V jednom provolání této metody se spočte jedna či více neznámých (záleží jestli je daný uzel supernodem). Struktura eliminačního stromu zajišťuje, že neznámé, které daný sloupec (uzel) ovlivňují jsou již spočítané. Po dosažení všech kořenů program přechází do části zpětné substituce.



Obrázek 10: Proces dopředné substituce pomocí eliminačního stromu ([3], upraveno)

Pro zpětnou substituci obsahuje třída *TreeNode* metodu *performBackwardSubstitution*, která se zavolá nad všemi kořeny (kořeny známe z procesu faktorizace). Rekurzivním způsobem je tato metoda provolávána na potomcích jednotlivých uzlů až k samotným listům. Na obrázku 11 vidíme, že v procesu zpětné substituce pracujeme s horní trojúhelníkovou maticí. Z toho důvodu je výsledný faktor před zpětnou substitucí transponován (originální faktor ale zůstává k dispozici). Po dosažení všech listů získáváme výsledek.



Obrázek 11: Proces zpětné substituce pomocí eliminačního stromu ([3], upraveno)

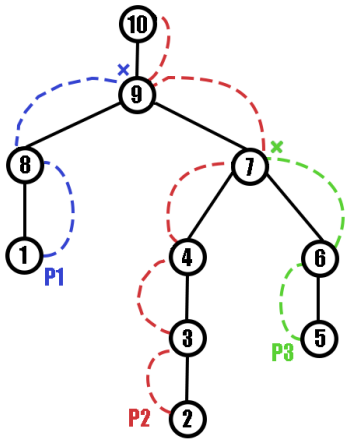
4.8 Paralelismus

Nezávislé zpracování jednotlivých větví eliminačního stromu vybízí k použití alespoň základního paralelismu. V programu využíváme soustavu direktiv *OpenMP*, přesněji direktivu pro paralelní cyklus.

```
#pragma omp parallel for
for (int i = 0; i < leaves.size(); i++) {
   leaves[i]->performElimination(result, AMatrix, roots);
}
```

Výpis 5: Spuštění paralelního cyklu pomocí OpenMP

Postup paralelního zpracování eliminačního stromu je popsán na obrázku 12. Proces $P1$ provede eliminaci nad uzly 1 a 8 a s velkou pravděpodobností dorazí jako první do uzlu číslo 9. Ten v daný moment ještě nebude mít k dispozici všechny potřebné update matice a tak proces $P1$ skončí. Mezitím procesy $P2$ a $P3$ zpracovávají své vlastní větve. Proces $P3$ s největší pravděpodobností jako první dorazí do uzlu číslo 7. Ten v daný moment opět nebude mít k dispozici všechny potřebné update matice a tak proces $P3$ skončí.



Obrázek 12: Paralelní zpracování eliminačního stromu

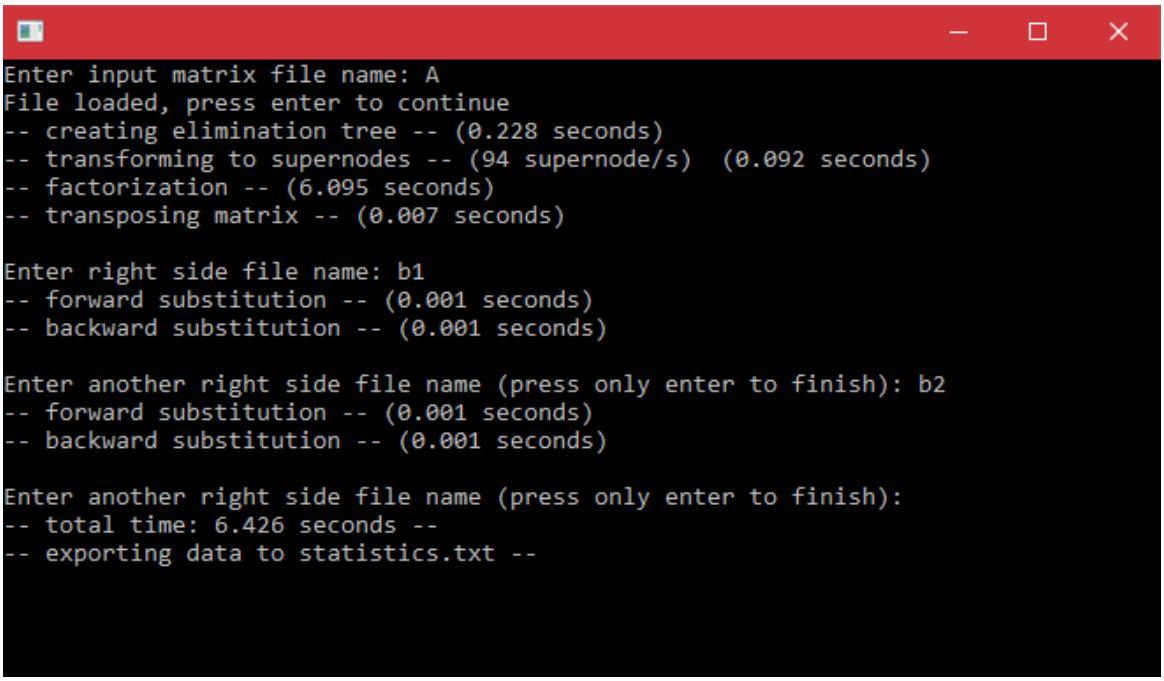
Jakmile se proces $P2$ dostane na uzel číslo 7, veškeré update matice budou k dispozici. Proces provede krok eliminace a bude pokračovat na uzel číslo 9. Tam nastane obdobná situace a proces se dostane až ke kořenu, čímž dokončí celou faktorizaci.

Všechny procesy zapisují eliminované sloupce do výsledné matice *result*. Aby nedošlo k přepisu dat, je tato část programu ošetřena direktivou *critical*, která slouží k řízení přístupu jednotlivých procesů.

Efektivita takto navrženého paralelismu záleží na tvaru eliminačního stromu (lesu). Obecně platí, že pro větší počet listů bude tento paralelismus efektivnější. Vytváření frontálních a update matic je náročnější, čím blíže jsme ke kořenu, takže při nevhodném tvaru eliminačního stromu nebude efektivita paralelismu ideální.

4.9 Ukázka programu a výstupních dat

Při použití program vypisuje jednotlivé kroky a naměřené časy (obrázek 13). Uživatel má možnost zadat více pravých stran, výsledek pro každou pravou stranu je po výpočtu exportován do souboru *_result.txt* (před podtržítko je přidán název souboru s pravou stranou). Po ukončení programu jsou statistické informace exportovány do souboru *statistics.txt*. Tento soubor obsahuje časy jednotlivých operací, počet listů, počet kořenů, počet supernodů a počet všech uzlů obsažených v supernodech (obrázek 14).



Obrázek 13: Ukázka konzolové aplikace

```
times:
creating elimination tree 1.824 seconds
transforming to supernodes 0.457 seconds
factorization 47.447 seconds
transposing matrix 0.046 seconds
forward substitution 0.003 seconds
backward substitution 0.003 seconds
total time: 49.78 seconds
roots: 81
leaves: 113
original nodes count: 2000
all supernodals: 1809
supernodes count: 144

-- end of one run --
```

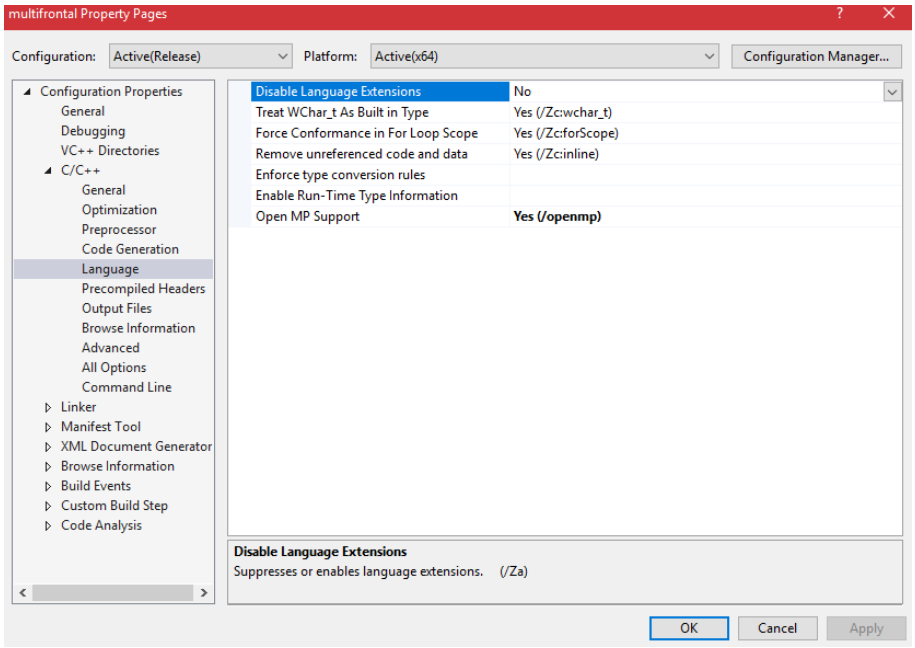
Obrázek 14: Ukázka výstupních statistických dat

4.10 Spuštění aplikace

Ke kompilaci a spuštění aplikace na operačním systému Linux použijeme následující dva příkazy:

```
g++ *.cpp -o app -fopenmp
./app
```

Na operačním systému Windows je doporučeno ke kompilaci a spuštění použít vhodné IDE (Integrated Development Environment), například Microsoft Visual Studio. Ve Visual Studiu je třeba v nastavení projektu povolit použití OpenMP (obrázek 15).



Obrázek 15: Povolení OpenMP ve Visual Studiu

5 Testování vlastní implementace

V této kapitole si popíšeme praktické otestování vlastní implementace. Testování proběhlo na dvou strojích:

- Stolní počítač - Intel Pentium G620 2.60 GHz (2 jádra), RAM 8GB DDR3, operační systém Windows 10 64-bit
- Notebook - Intel Core i5-3337U 1.80 GHz (4 jádra), RAM 4GB DDR3, operační systém Windows 8 64-bit

Data použita k testování byla vygenerována v MATLABu. Ke vygenerování pozitivně definitních symetrických matic byla použita funkce *sprandsym*. Vstupními parametry této funkce jsou rozměr generované matice a hustota rozložení nenulových prvků. k vygenerování pravé strany byla použita funkce *rand*. Tato funkce generuje vektor uniformně rozložených náhodných čísel z intervalu $(0, 1)$. Parametry funkce jsou rozměry generovaného vektoru. Vygenerovaná data byla následně upravena do správného formátu a exportována do textových souborů. Tyto soubory jsou již plně kompatibilní se samotnou aplikací.

Vzhledem k výpočetní náročnosti funkce *sprandsym* při generování větších matic byla jako největší matice zvolena matice s rozměrem $n = 3000$ a hustotou výskytu nenulových prvků 10%. Při jednotlivých měřeních pak byly použity matice s $n = 1000$, 2000 a 3000 a hustotami výskytu nenulových prvků 10%, 5% a $\frac{100}{n}$% (nenulové prvky pouze na hlavní diagonále). Při použití hustoty $\frac{100}{n}$% je generování matic rychlé a proto byly tyto matice otestovány s vyšším počtem neznámých (60 tisíc a více).

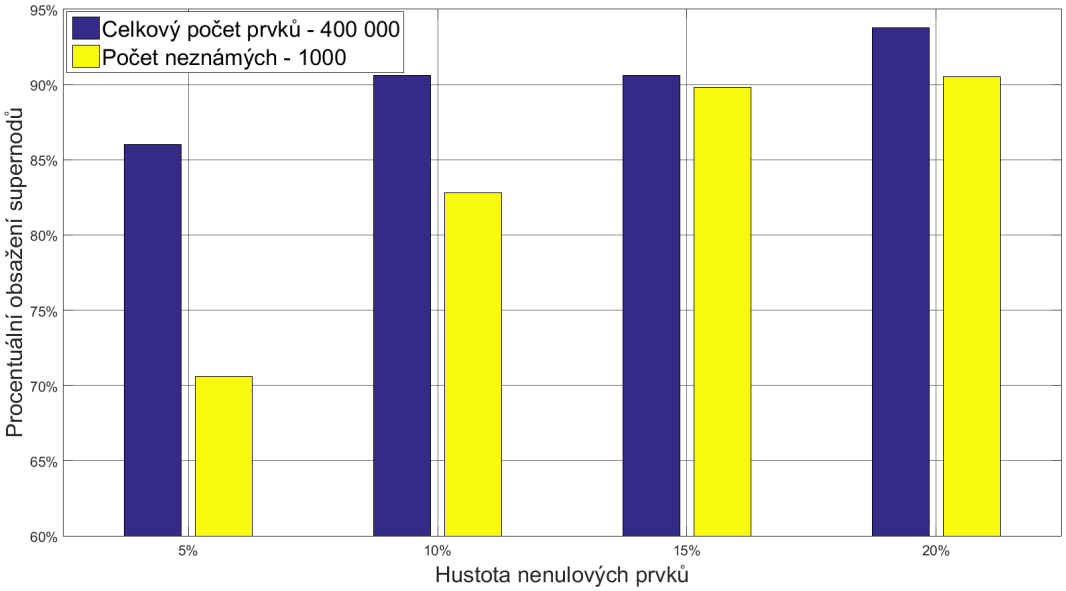
5.1 Výskyt supernodů

Supernody zmíněné v 3.6 hrají důležitou roli ve vylepšení multifrontální metody. Struktura matice A ovšem může obecně vést k jejich nulovému výskytu. Data která byla použita sice nereprezentují žádnou reálnou situaci, ale i náhodně vygenerovaná data mají vypovídající hodnotu o četnosti výskytu supernodů. Na obrázku 16, můžeme vidět procentuální obsažení supernodálních uzlů. Byly porovnány dva případy:

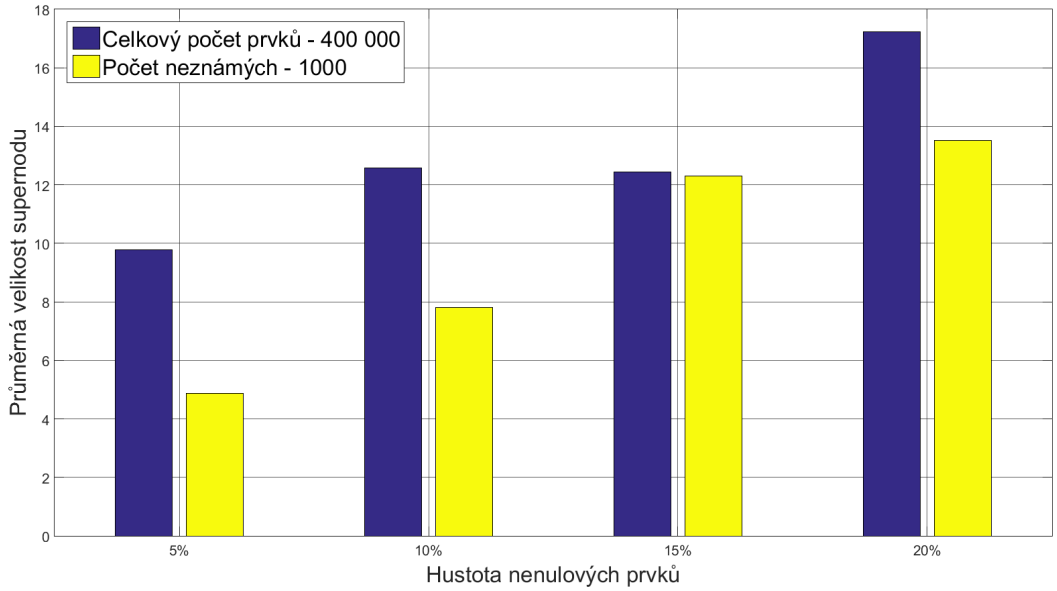
1. Stejný počet neznámých (1000), ale různý počet nenulových prvků (žlutý sloupec)
2. Stejný počet nenulových prvků, ale různý počet neznámých (modrý sloupec)

Z grafu vyplývá, že při stejném počtu všech nenulových prvků je výskyt supernodů zhruba stejný a na hustotě výskytu nenulových prvků tolik nezáleží. Pokud porovnáváme matice se stejným počtem neznámých a rozdílnou hustotou, můžeme vidět, že s rostoucí hustotou roste i počet supernodů, což ostatně nepřímo vyplývá z předchozího poznatku.

Stejným způsobem byla porovnána i průměrná velikost jednoho supernodu (obrázek 17). Velikostí se uvažuje počet klasických uzlů obsažených v jednom supernodu. Z grafu vyplývá, že průměrná velikost supernodu roste s vyšší hustotou výskytu nenulových prvků.



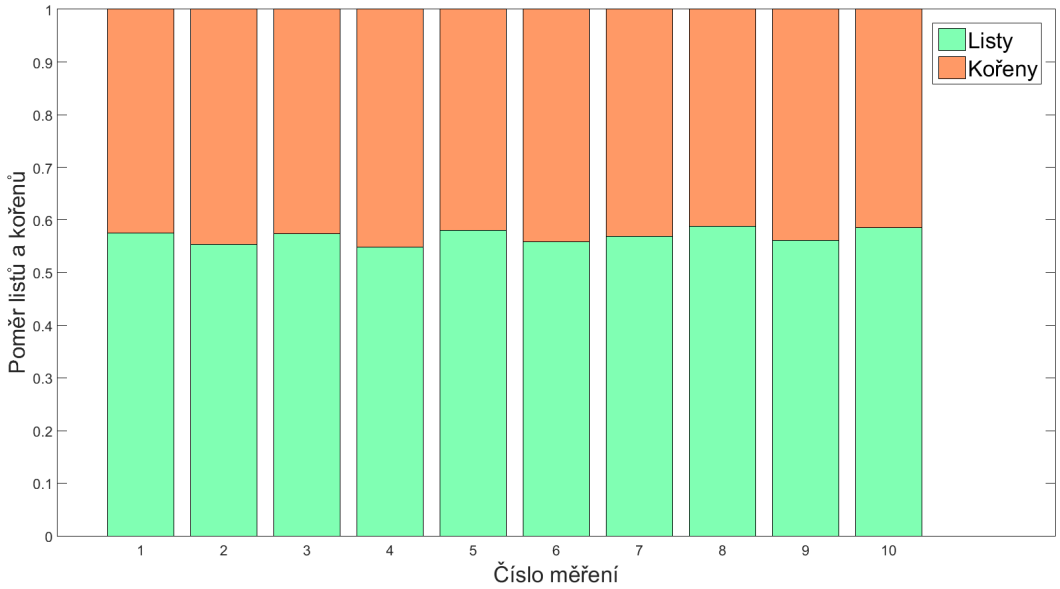
Obrázek 16: Procentuální obsažení supernodů



Obrázek 17: Průměrná velikost supernodu

5.2 Listy a kořeny

V 4.8 jsme hovořili o efektivnosti „naivního“ paralelismu nad eliminačním stromem. Ta je závislá na struktuře eliminačního stromu - počet listů a rozvětvení, délka větví, počet kořenů, apod. Nejjednodušším způsobem, jak získat představu o struktuře eliminačního stromu, je zjištění poměru mezi listy a kořeny. Tento poměr je vyobrazen na obrázku 18. Obecně bychom předpokládali znatelně vyšší výskyt listů, ale jak můžeme vidět z grafu, poměr listů a kořenů je celkem vysoký. Tato skutečnost má pozitivní vliv na náš „naivní“ paralelismus.



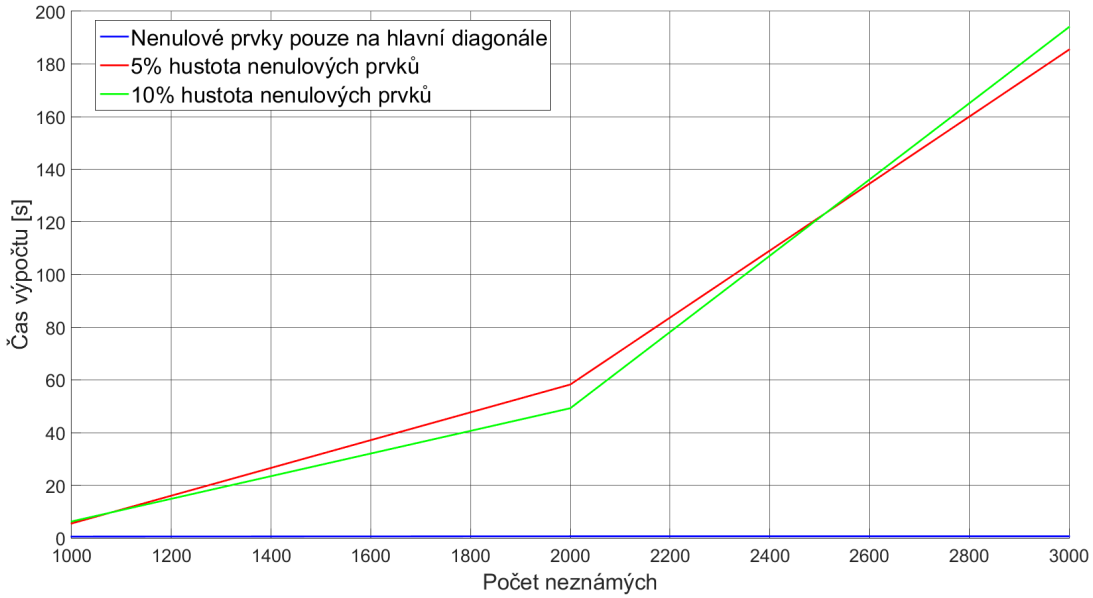
Obrázek 18: Poměr listů a kořenů

5.3 Výkon aplikace

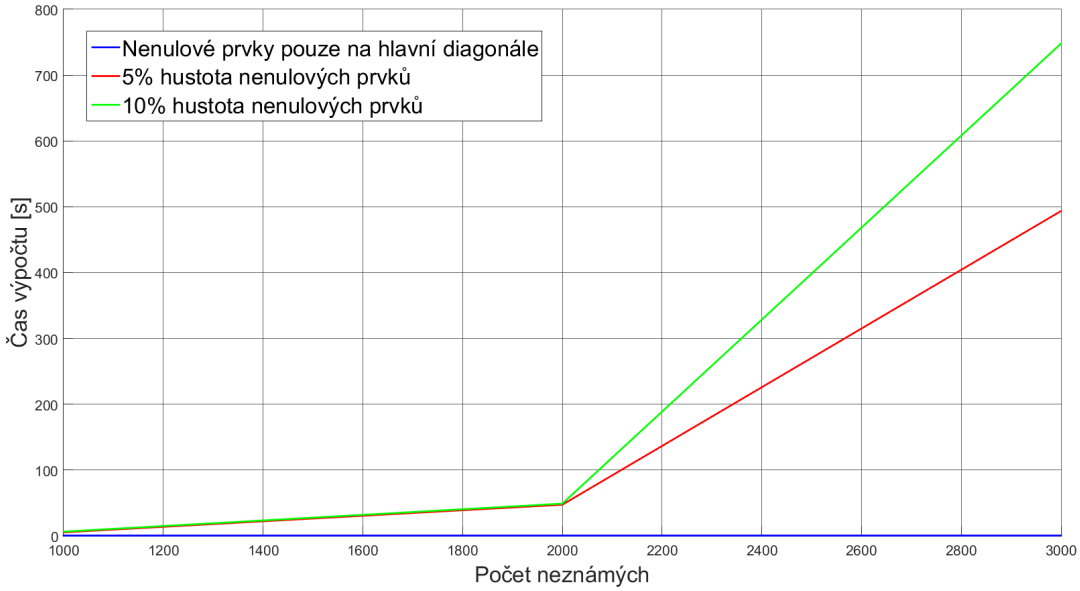
I přestože hlavním cílem aplikace byl kvalitní a přehledný kód, podívejme se na její výkon. Měření probíhalo na maticích o hustotách výskytu nenulových prvků 10%, 5% a $\frac{100}{n}$%. S rostoucím počtem prvků rapidně klesala rychlost výpočtu. Nejdražší operací aplikace je formování frontálních a update matic. Při vyšším počtu nenulových prvků potřebuje tato operace velké množství operační paměti a výkonu. Notebook, který má vyšší počet jader prováděl výpočty v horším čase než stolní počítač. Tato skutečnost je zapříčiněna malou RAM pamětí a naivitou paralelismu. Pří formování velkých frontálních a update matic aplikace požadovala i 4GB operační paměti. Notebook v tomto případě musel uchovávat data na pevném disku, což je oproti RAM řádově pomalejší. Notebook byl pomalejší i v případě, kdy se nenulové prvky vyskytovaly pouze na hlavní diagonále (obrázek 21). V tomto případě je eliminační strom tvořen pouze z listů, nad kterými jsou formovány malé frontální matice. Při takovéto struktuře eliminačního stromu a velikostí frontálních matic by měl získat výhodu méně výkonný procesor s vyšším počtem

jader. Naměřené hodnoty ovšem ukazují, že výkonnější procesor vyhrává i v této situaci. Pokud porovnáme verzi bez supernodů se supernodální, zjistíme, že supernodální verze je daleko rychlejší. Z programátorského hlediska je to zapříčiněno menším generováním instancí jednotlivých objektů - supernodální verze pracuje v jednom kroku s více sloupci, které ale zpracovává nad jednou frontální maticí. U verze bez supernodů může být ve stejném kroku vytvořených více frontálních matic.

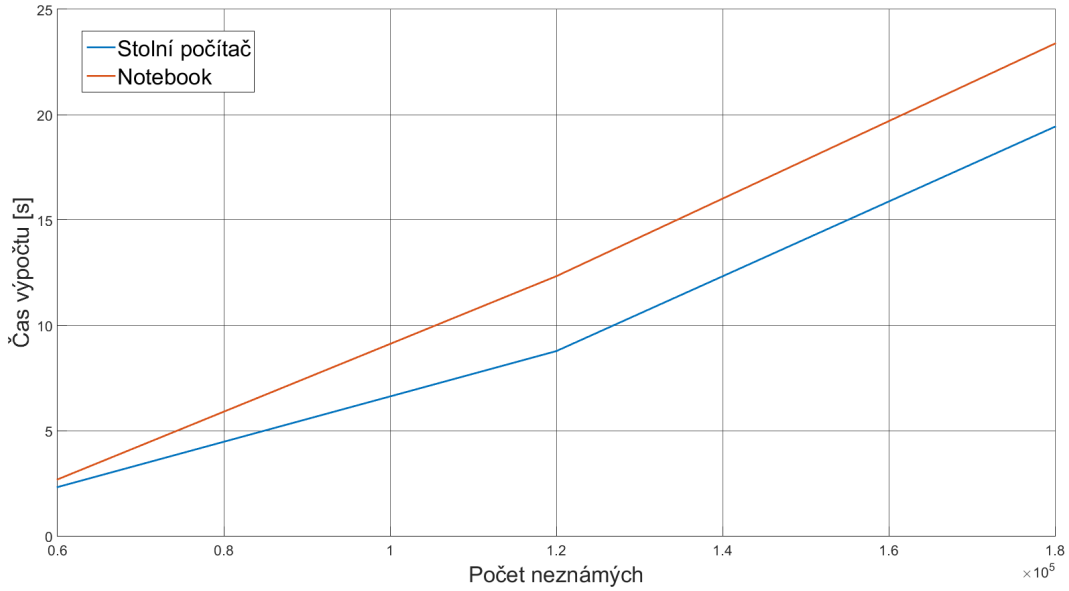
Existující implementace multifrontální metody jsou daleko rychlejší a srovnání z hlediska výkonu nemá v tomto případě smysl.



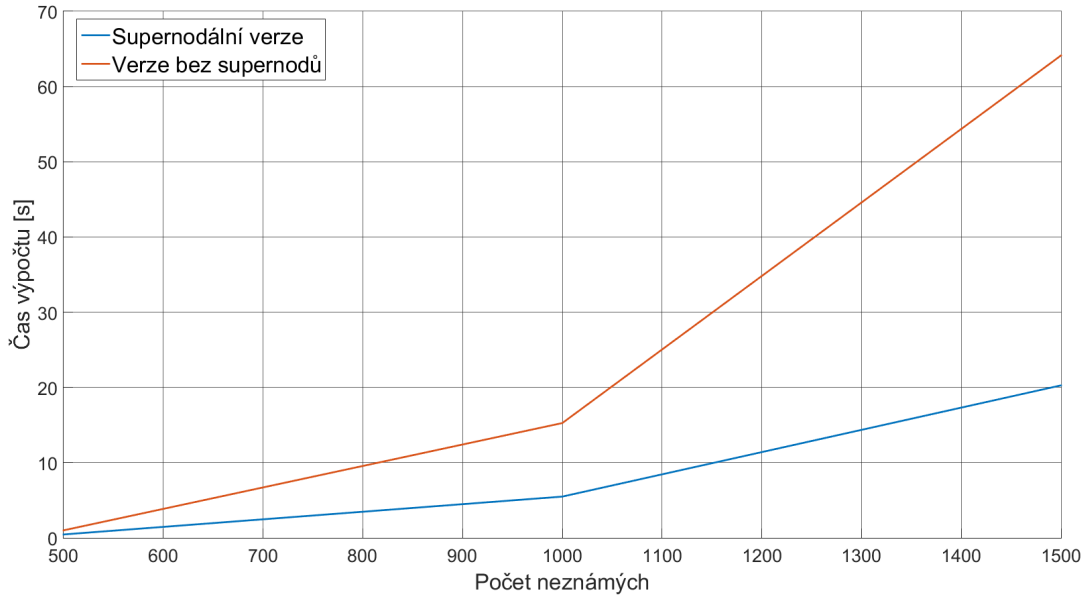
Obrázek 19: Výkon aplikace změřen na stolním PC



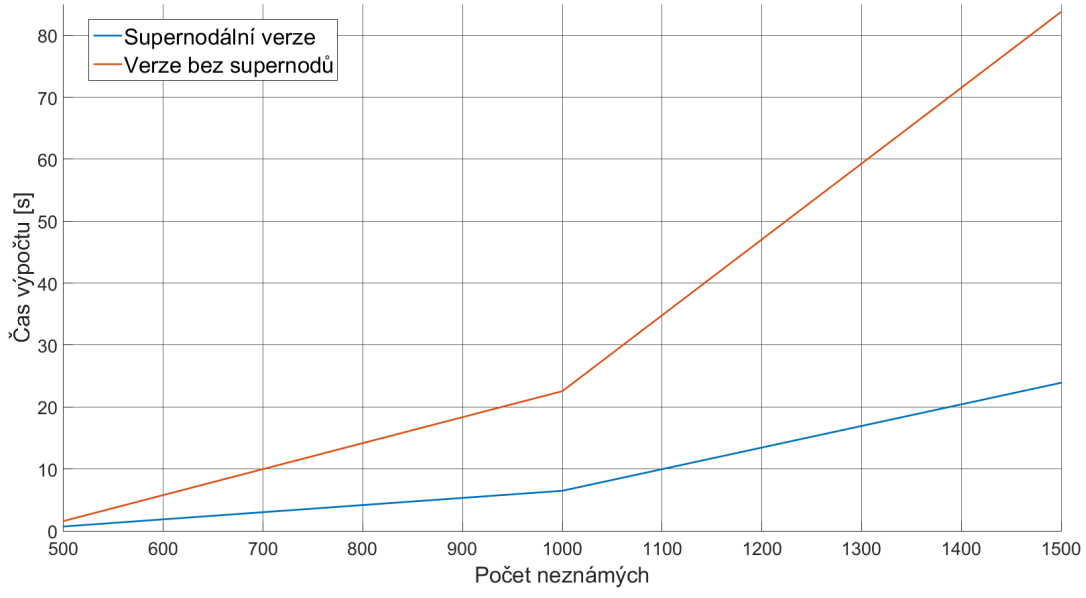
Obrázek 20: Výkon aplikace změřen na notebooku



Obrázek 21: Výkon aplikace pří výskytu nenulových prvků pouze na hlavní diagonále



Obrázek 22: Porovnání supernodální verze s klasickou, 5% hustota výskytu nenulových prvků



Obrázek 23: Porovnání supernodální verze s klasickou, 10% hustota výskytu nenulových prvků

6 Existující přímé solvery

Multifrontální metoda byla vytvořena v 80. letech 20. století. Od té doby vzniklo několik knihoven, které multifrontální metodu využívají k řešení soustav lineárních rovnic. Většina těchto implementací je napsaná v programovacím jazyce C nebo Fortran a jejich cílem je být co nejefektivnější. V této kapitole si některé tyto knihovny představíme.

6.1 BLAS

BLAS[6] (Basic Linear Algebra Subprograms) je soubor nízkoúrovňových rutin určených k provádění běžných lineárních algebraických operací. Mezi tyto operace patří násobení matic a vektorů, skalární součin, násobení skalárem apod. Tyto operace lze rozdělit do tří úrovní:

- Level 1 BLAS provádí skalární, vektorové a vektoro-vektorové operace
- Level 2 BLAS provádí matico-vektorové operace
- Level 3 BLAS matico-maticové operace

Všechny tyto úrovně jsou rozděleny do čtyř kategorií - single precision, double precision, complex precision a double complex precision. Nejsilnější stránkou BLASu je jeho výkonnost - existuje řada verzí specifických pro dané zařízení a jeho architekturu. Tyto optimalizované verze jsou k dispozici u daných prodejců (Intel, AMD a další). BLAS je pro svůj výkon, přenositelnost a širokou dostupnost používán u vývoje kvalitního algebraického softwaru. BLAS sám o sobě multifrontální metodu neimplementuje, ale jeho rutiny jsou k tomu používány.

6.2 MUMPS

MUMPS[9] (Multifrontal Massively Parallel sparse direct Solver) je balík rutin pro řešení systémů lineárních rovnic ve tvaru $Ax = b$, kde A je čtvercová řídká matice, která může být nesymetrická, symetrická definitně pozitivní nebo obecně symetrická. Knihovna je založena na volání jedné rutiny, která rozhoduje o aktuálním kroku na základě tzv. JOBu. V ukázce 6 má tato rutina název *dmumps_c()*. Struktura *id* v sobě nese veškeré potřebné informace včetně zmíněno JOBu. Hodnoty, kterých JOB může nabývat jsou:

1. Analýza - uspořádání a symbolická faktorizace
2. Škálování a numerická faktorizace
3. Fáze řešení a analýza chyb
4. Analýza následována numerickou faktorizací
5. Analýza, numerická faktorizace a fáze řešení

Operace prováděné na základě hodnoty JOBu jsou mezi sebou nezávislé (záleží na uživateli kterou operaci v danou chvíli potřebuje).

```
#include "dmumps_c.h"
DMUMPS_STRUC_C id;
// definice prave strany a matice A
rhs[0]=1.0;rhs[1]=4.0;
a[0]=1.0;a[1]=2.0;
// inicializace MUMPS instance
id.comm_fortran=USE_COMM_WORLD;
id.par=1; id.sym=0;
id.job=JOB_INIT;
dmumps_c(&id);
// definice problemu na hostitelskm procesu
if(myid == 0) {
   id.n = n; id.nnz =nnz; id.irn=irn; id.jcn=jcn;
   id.a = a; id.rhs = rhs;
   }
// zavolani MUMPS baliku (analyza, faktorizace a vyreseni)
id.job=5;
dmumps_c(&id);
// vycisteni MUMPS instance
id.job=JOB_END;
dmumps_c(&id);
```

Výpis 6: Ukázka řešení systému rovnic pomocí knihovny MUMPS

6.3 UMFPACK

UMFPACK je soubor rutin určených k řešení nesymetrických řídkých systémů lineárních rovnic pomocí nesymetrické multifrontální metody. Knihovna je součástí balíčku SuiteSparse[8] a vytvořil ji Timothy Davis. UMFPACK je k dispozici v C, C++, Fortran a MATLAB verzi. Z BLAS rutin využívá UMFPACK rutiny *DGEMM* a *DSYMM*, které slouží k násobení hustých matic. Jako vstupní data UMFPACK vyžaduje matici v CSC formátu (zapsanou ve třech obyčejných polích) a vektor pravé strany (taktéž obyčejné pole). V 7 je popsáno volání jednotlivých rutin pro vyřešení zadaného systému rovnic.

```
#include "umfpack.h"
// pripava dat
int n = 5 ;
int Ap [ ] = {0, 2, 5, 9, 10, 12};
```

```
int Ai [ ] = {0, 1, 0,  2,  4,  1,  2,  3,  4,  2,  1,  4};
double Ax [ ] = {2., 3., 3., -1., 4., 4., -3., 1., 2., 2., 6., 1.};
double b [ ] = {8., 45., -3., 3., 19.};
double x [5];
double *null = (double *) NULL;
void *Symbolic, *Numeric;
// symbolicka faktorizace
umfpack_di_symbolic (n, n, Ap, Ai, Ax, &Symbolic, null, null );
// numericka faktorizace
umfpack_di_numeric (Ap, Ai, Ax, Symbolic, &Numeric, null, null );
// uvolneni pameti asociovane se symbolickou faktorizaci
umfpack_di_free_symbolic (&Symbolic );
// vyreseni linearniho systemu
umfpack_di_solve (UMFPACK_A, Ap, Ai, Ax, x, b, Numeric, null, null );
// uvolneni pameti asociovane s numerickou faktorizaci
umfpack_di_free_numeric (&Numeric );
```

Výpis 7: Ukázka řešení systému rovnic pomocí knihovny UMFPACK

6.4 SuperLU

SuperLU[7] (Supernodal LU) je univerzální knihovna pro řešení nesymetrických řídkých systémů lineárních rovnic. Knihovna je napsána v jazyce C a je možné ji volat z jazyků C a Fortran. SuperLU využívá k podpoře paralelismu technologie MPI, OpenMP a CUDA. Rutiny knihovny provádějí LU rozklad s částečným výběrem hlavního prvku. Z BLAS rutin využívá SuperLU rutiny *DGEMM* a *DSYMM*, které slouží k násobení hustých matic a rutiny *DGEMV* a *DSYMV* sloužící k násobení vektoru a matice. SuperLU požaduje na vstupu matici v CSC formátu (zapsanou ve třech obyčejných polích) a vektor pravé strany (taktéž obyčejné pole). V 8 je popsáno volání jednotlivých rutin pro vyřešení zadaného systému rovnic.

```
#include "slu_ddefs.h"
// pripava dat
SuperMatrix A, L, U, B;
superlu_options_t options;
SuperLUStat_t stat;
int m = 5;
int n = 5;
int nnz = 12;
int *perm_r = intMalloc(m))
int *perm_c = intMalloc(n))
```

```
int xa [ ] = {0, 2, 5, 9, 10, 12};
int asub [ ] = {0, 1, 0,  2,  4,  1,  2,  3,   4,  2,  1,  4};
double a [ ] = {2., 3., 3., -1., 4., 4., -3., 1., 2., 2., 6., 1.};
double rhs [ ] = {8., 45., -3., 3., 19.};
// vytvoreni matice A ve formatu ocekavanem SuperLU
dCreate_CompCol_Matrix(&A, m, n, nnz, a, asub, xa, SLU_NC, SLU_D, SLU_GE);
// vytvoreni matice prave strany
dCreate_Dense_Matrix(&B, m, 1, rhs, m, SLU_DN, SLU_D, SLU_GE);
// nastaveni defaultnich vstupnich hodnot
set_default_options(&options);
options.ColPerm = NATURAL;
// inicializace statickych promennych
StatInit(&stat);
// vyreseni linearniho systemu
dgssv(&options, &A, perm_c, perm_r, &L, &U, &B, &stat, &info);
// uvolneni pameti
SUPERLU_FREE (rhs);
SUPERLU_FREE (perm_r);
SUPERLU_FREE (perm_c);
Destroy_CompCol_Matrix(&A);
Destroy_SuperMatrix_Store(&B);
Destroy_SuperNode_Matrix(&L);
Destroy_CompCol_Matrix(&U);
StatFree(&stat);
```

Výpis 8: Ukázka řešení systému rovnic pomocí knihovny SuperLU

7 Závěr

Hlavním cílem práce bylo seznámení se s multifrontální metodou a její následná implementace a otestování. Tento cíl byl splněn. Multifrontální metoda byla implementována objektovým způsobem v jazyce C++ a otestována na vybraných počítačích.

V teoretické části byly popsány pojmy nutné k pochopení multifrontální metody (a přímých metod). Část zabývající se praktickou implementací obsahovala popis struktury aplikace a řešení problémů, které při implementaci vyvstaly. Zmíněny byly i některé programátorské konvence (práce s pamětí v jazyce C++). V následující části proběhlo otestování aplikace na datech vygenerovaných v MATLABu. Testování pro všechna testovací data proběhlo v pořádku. Byla změřena rychlost aplikace a analyzováno její ovlivnění strukturou vstupních dat. Rychlost aplikace byla oproti existujícím knihovnám řádově pomalejší. Tento výsledek byl očekávaný, při optimalizaci aplikace by musel být opuštěn objektový přístup a aplikace by musela být vyvíjena s jiným cílem. V poslední části byly představeny existující knihovny pro řešení řídkých systémů lineárních rovnic.

Literatura

[1] LIU, Joseph W. H. THE MULTIFRONTAL METHOD FOR SPARSE MATRIX SOLUTION: THEORY AND PRACTICE [online]. 1992. s. 82-109 [cit. 2017-04-26]

[2] van GRONDELLE, Jeroen. Symbolic Sparse Cholesky Factorisation Using EliminationTtrees [online]. 1999. [cit. 2017-04-26]

[3] AMESTOY, P. a A. BUTTARI. Sparse Linear Algebra: Direct methods, advanced features [online]. [cit. 2017-04-26]

[4] AMESTOY, Patrick a Chiara PUGLISI. An unsymmetrized multifrontal LU factorization [online]. 2000. [cit. 2017-04-26].

[5] DAVIS, Timothy. Direct Methods for Sprase Linear Systems. 2006. ISBN 9780898176139.

[6] BLAS (Basic Linear Algebra Subprograms) [online]. [cit. 2017-04-26]. Dostupné z: http://www.netlib.org/blas/

[7] SuperLU [online]. [cit. 2017-04-26]. Dostupné z: http://crd-legacy.lbl.gov/xiaoye/SuperLU/

[8] SuiteSparse: a suite of sparse matrix software [online]. [cit. 2017-04-26]. Dostupné z: http://faculty.cse.tamu.edu/davis/suitesparse.html

[9] MUMPS: a parallel sparse direct solver [online]. [cit. 2017-04-26]. Dostupné z: http://mumps.enseeiht.fr/